Macintosh
Technical Notes

Developer Technical Support

Index

June 1992

Note: Trap names can appear in two different places: under the name of the trap alphabetically and at the end of the index, preceeded by an underscore (_).

Macintosh
Technical Notes

Developer Technical Support

#0: About Macintosh Technical Notes June 1992

Technical Note #0 (this document) accompanies each release of Macintosh Technical Notes. This release includes new Notes 317, 318, and an index to all released Macintosh Technical Notes. If there are any subjects which you would like to see treated in a Technical Note (or if you have any questions about existing Technical Notes), please contact us at one of the following addresses:

Macintosh Technical Notes
Developer Technical Support
Apple Computer, Inc.
20525 Mariani Avenue, M/S 75-3T
Cupertino, CA 95014
AppleLink: MacDTS
MCI Mail: MacDTS
Internet: MacDTS@AppleLink.Apple.com

We want Technical Notes to be distributed as widely as possible, so they are sent to all Partners and Associates at no charge; they are also posted on AppleLink in the Developer Services bulletin board and other electronic sources, including the Apple FTP site (IP 130.43.2.3). You can also order them through APDA. As an APDA customer, you have access to the tools and documentation necessary to develop Apple-compatible products. For more information about APDA, contact:

APDA
Apple Computer, Inc.
20525 Mariani Avenue, M/S 33-G
Cupertino, CA 95014
(800) 282-APDA or (800) 282-2732
Fax: (408) 562-3971
Telex: 171-576
AppleLink: APDA
Internet: APDA@AppleLink.Apple.com

We place no restrictions on copying Technical Notes, with the exception that you cannot resell them, so read, enjoy, and share. We hope Macintosh Technical Notes will provide you with lots of valuable information while you are developing Macintosh hardware and software. The following pages list all Macintosh Technical Notes that have been released (both by number and by subject).

Released Macintosh Technical Notes

June 1992

Indexed by Number

New ***
Revised *R*

Number	Title	Released
1	Desk Accessories and System Resources	obsolete 3/88
2	Compatibility Guidelines	3/88
3	Command-Shift-Number Keys	3/88
4	Error Returns from GetNewDialog	3/88
5	Using Modeless Dialogs from Desk Accessories	3/88
6	Shortcut for Owned Resources	3/88
7	A Few Quick Debugging Tips	3/88
8	RecoverHandle Bug in AppleTalk Pascal Interfaces	obsolete 3/88
9	Will Your AppleTalk Application Support Internets?	4/90
10	Pinouts	3/88
11	Memory-Based MacWrite Format	obsolete 8/89
12	Disk-Based MacWrite Format	obsolete 8/89
13	MacWrite Clipboard Format	obsolete 8/89
14	The INIT 31 Mechanism	obsolete 3/88
15	Finder 4.1	obsolete 3/88
16	MacWorks XL	obsolete 3/88
17	Low-Level Print Driver Calls	obsolete 3/88
18	TextEdit Conversion Utility	3/88
19	How to Produce Continuous Sound Without Clicking	6/89
20	Data Servers on Appletalk	3/88
21	QuickDraw's Internal Picture Definition	3/88
22	TEScroll Bug	3/88
23	Life With Font/DA Mover—Desk Accessories	3/88
24	Available Volumes	3/88
25	Don't Depend on Register A5 Within Trap Patches	3/88
26	Fond of FONDS	5/92
27	MacDraw 'PICT' File Format	obsolete 8/89
28	Finders and Foreign Drives	3/88
29	Resources Contained in the Desktop File	3/88
30	Font Height Tables	3/88
31		unused
32	Reserved Resource Types	3/88
33	ImageWriter II Paper Motion	3/88
34	User Items in Dialogs	10/88
35	DrawPicture Problem	obsolete 3/88
36	Drive Queue Elements	3/88
37	Differentiating Between Logic Boards	obsolete 3/88
38	The ROM Debugger	3/88
39	Segment Loader Patch	obsolete 3/88
40	Finder Flags	3/88
41	Drawing Into an Off-Screen Bitmap	6/90
42	Pascal Routines Passed by Pointer	3/88
43	Calling LoadSeg	obsolete 3/88
44	HFS Compatibility	3/88
45	*Inside Macintosh* Quick Reference	obsolete 3/88
46	Separate Resource Files	3/88
47	Customizing Standard File	3/88
48	Bundles	3/88
49		unused

50	Calling SetResLoad	3/88
51	Debugging With PurgeMem and CompactMem	3/88
52	Calling Launch From a High-Level Language	obsolete 4/89
53	MoreMasters Revisited	3/88
54	Limit to Size of Resources	obsolete 3/88
55	Drawing Icons	3/88
56	Break/CTS Device Driver Event Structure	3/88
57	Macintosh Plus Overview	obsolete 3/88
58	International Utilities Bug	obsolete 3/88
59	Pictures and Clip Regions	3/88
60	Drawing Characters in a Narrow GrafPort	3/88
61	GetItemStyle Bug	obsolete 3/88
62	Don't Use Resource Header Application Bytes	3/88
63	WriteResource Bug Patch	obsolete 3/88
64	IAZNotify	obsolete 3/88
65	Macintosh Plus Pinouts	3/88
66	Determining Which File System is Active	8/90
67	Finding the "Blessed Folder"	5/92
68	Searching Volumes-Solutions and Problems	1/92
69	Setting ioFDirIndex in PBGetCatInfo Calls	3/88
70	Forcing Disks to be Either 400K or 800K	3/88
71	Finding Drivers in the Unit Table	3/88
72	Optimizing for the LaserWriter—Techniques	10/90
73	Color Printing	3/88
74	Don't Use the Resource Fork for Data	3/88
75	Apple's Multidisk Installer	1/92
76	The Macintosh Plus Update Installation Script	obsolete 3/88
77	HFS Ruminations	3/88
78	Resource Manager Tips	3/88
79	_ZoomWindow	4/90
80	Standard File Tips	3/88
81	Caching	3/88
82	TextEdit: Advice & Descent	3/88
83	System Heap Size Warning	3/88
84	Edit File Format	3/88
85	GetNextEvent; Blinking Apple Menu	3/88
86	MacPaint Document Format	6/89
87	Error in FCBPBRec	3/88
88	Signals	3/88
89	DrawPicture Bug	obsolete 3/88
90	SANE Incompatibilities	obsolete 3/88
91	Optimizing for the LaserWriter—Picture Comments	3/88
92	The Appearance of Text	3/88
93	MPW: {$LOAD} ;_DataInit; %_MethTables	3/88
94	Tags	3/88
95	How to Add Items to the Print Dialogs	3/88
96	SCSI Bugs	3/88
97	PrSetError Problem	obsolete 3/88
98	Short-Circuit Booleans in Lisa Pascal	obsolete 3/88
99	Standard File Bug in System 3.2	obsolete 3/88
100	Compatibility with Large-Screen Displays	3/88
101	CreateResFile and the Poor Man's Search Path	3/88
102	HFS Elucidations	3/88
103	Using MaxApplZone and MoveHHi from Assembly	3/88
104	MPW: Accessing Globals From Assembly Language	3/88

105	MPW Object Pascal Without MacApp	3/88
106	The Real Story: VCBs and Drive Numbers	3/88
107	Nulls in Filenames	3/88
108	_AddDrive, _DrvrInstall, and _DrvrRemove	12/88
109	Bug in MPW 1.0 Language Libraries	obsolete 3/88
110	MPW: Writing Stand-Alone Code	obsolete 8/90
111	MoveHHi and SetResPurge	3/88
112	FindDItem	3/88
113	Boot Blocks	3/88
114	AppleShare and Old Finders	3/88
115	Application Configuration with Stationery Pads	3/88
116	AppleShare-able Apps. and the Resource Manager	3/88
117	Compatibility: Why and How	3/88
118	How to Check and Handle Printing Errors	obsolete 10/90
119	Determining if Color QuickDraw Exists	obsolete 3/88
120	Principia Off-Screen Graphics Environments	3/92
121	Using the High-Level AppleTalk Routines	3/88
122	Device-Independent Printing	3/88
123	Bugs in LaserWriter ROMs	3/88
124	Using Low-Level Printing Calls With AT ImageWriters	3/88
125	Effect of Spool-a-page/Print-a-page on Shared Printers	3/88
126	Sub(Launching) from a High-Level Language	4/89
127	TextEdit EOL Ambiguity	3/88
128	PrGeneral	3/88
129	_Gestalt & _SysEnvirons—A Never-Ending Story	5/92
130	Clearing ioCompletion	3/88
131	TextEdit Bugs in System 4.2	3/88
132	AppleTalk Interface Update	3/88
133	Am I Talking to a LaserShare Spooler?	3/88
134	Hard Disk Medic & Booting Camp	3/88
135	Getting through CUSToms	3/88
136	Register A5 Within GrowZone Functions	3/88
137	AppleShare 1.1 Server FPMove Bug	3/88
138	Using KanjiTalk with a non-Japanese Macintosh Plus	3/88
139	Macintosh Plus ROM Versions	3/88
140	Why PBHSetVol is Dangerous	3/88
141	Maximum Number of Resources in a File	3/88
142	Avoid Use of Network Events	3/88
143	Don't Call ADBReInit on the SE with System 4.1	3/88
144	Macintosh Color Monitor Connections	2/91
145	Debugger FKEY	obsolete 8/90
146	Notes on MPW's –mc68881 Option	5/92
147	Finder Notes: "Get Info" Default & Icon Masks	3/88
148	Suppliers for Macintosh II Board Developers	3/88
149	Document Names and the Printing Manager	3/88
150	Macintosh SE Disk Driver Bug	obsolete 3/88
151	System Error 33, "zcbFree has gone negative"	3/88
152	Using Laser Prep Routines	3/88
153	Changes in International Utilities and Resources	3/88
154	Displaying Large PICT Files	3/88
155	Handles and Pointers—Identity Crisis	3/88
156	Checking for Specific Functionality	3/88
157	Problem with GetVInfo	3/88
158	Frequently Asked MultiFinder Questions	3/88
159	Hard Disk Hacking	3/88

160	Key Mapping	2/91
161	A Printing Loop That Cares…	10/90
162	MPW 2.0 Pascal Compiler Bug	obsolete 3/88
163	Adding Color With CopyBits	3/88
164	MPW C Functions: To declare or not to declare, …	3/88
165	Creating Files Inside an AppleShare Drop Folder	3/88
166	MPW C Functions Using Strings or Points as Arguments	3/88
167	AppleShare Foreground Applications	3/88
168	HyperCard And You: Economy Edition	2/91
169	HyperCard 1.0.1 And 1.1 Anomalies	obsolete 2/91
170	HyperCard File Format	obsolete 2/91
171	Things You Wanted to Know About _PackBits*	1/92
172	Parameters for MDEF Message #3	3/88
173	PrGeneral Bug	3/88
174	Accessing the Script Manager Print Action Routine	3/88
175	SetLineWidth Revealed	3/88
176	Macintosh Memory Configurations	4/92
177	Problem with WaitNextEvent in MultiFinder 1.0	3/88
178	Modifying the Standard String Comparison	3/88
179	Setting ioNamePtr in File Manager Calls	3/88
180	MultiFinder Miscellanea	8/89
181	Every Picture [Comment] Tells Its Story, Don't it?	3/88
182	How to Construct Word-Break Tables	3/88
183	Position-Independent PostScript	3/88
184	Notification Manager	12/89
185	OpenRFPerm: What your mother never told you	4/88
186	Lock, Unlock the Range	2/91
187	Don't Look at ioPosOffset	4/88
188	ChangedResource: Too much of a good thing	4/88
189	Version Territory	10/90
190	Working Directories and MultiFinder	4/88
191	Font Names	8/88
192	Surprises in LaserWriter 5.0 and Newer	2/90
193	So Many Bitmaps, So Little Time	12/89
194	WMgrPortability	3/88
195	ASP and AFP Description Discrepancies	8/88
196	CDEF Parameters	10/89
197	Chooser Enhancements	8/88
198	Font/DA Mover, Styled Fonts, and NFNTs	5/92
199	KillNBP Clarification	8/88
200	MPW 2.0.2 Bugs	10/88
201	ReadPacket Clarification	8/88
202	Resetting the Event Mask	12/88
203	Don't Abuse the Managers	8/88
204	HFS Tidbits	8/88
205	MultiFinder Revisited, The 6.0 System Release	4/90
206	Space Aliens Ate My Mouse (ADB–The Untold Story)	10/91
207	Styled TextEdit Changes in System 6.0	12/88
208	Setting and Restoring A5	6/89
209	High Sierra & ISO 9660 CD ROM Formats	8/88
210	The Desktop File's Outer Limits	8/88
211	Palette Manager Changes in System 6.0.2	10/88
212	The Joy of Being 32-Bit Clean	8/91
213	_StripAddress: The Untold Story	8/90
214	New Resource Manager Calls	10/88

215	“New” cdev Messages	10/88
216	AppleShare 1.1 and 2.0 Limits	10/88
217	Where Have My Font Icons Gone?	4/91
218	New High-Level File Manager Calls	12/88
219	New Memory Manager Glue Routines	12/88
220	Segment Loader Limitations	12/88
221	NuBus Interrupt Latency (I Was a Teenage DMA Junkie)	10/89
222	Custom Menu Flashing Bug	2/89
223	Assembly Language Use of _InitGraf with MPW	2/89
224	Opening AppleTalk	2/89
225	Using RegisterName	2/89
226	Moving Your Cat	2/91
227	Toolbox Karma	2/89
228	Use Care When Swapping MMU Mode	2/90
229	A/UX 2.0 Compatibility Guidelines	2/91
230	Pertinent Information About the Macintosh SE/30	6/89
231	Macintosh Allegro Common Lisp Features	2/90
232	Strip With _OpenResFile and _OpenRFPerm	4/89
233	MultiFinder and _SetGrowZone	6/89
234	NuBus Physical Designs—Beware	12/89
235	Cooperating with the Coprocessor	10/90
236	Speedy the Math Coprocessor	6/89
237	TextEdit Record Size Limitations Revisited	6/89
238	Getting a Full Pathname	10/89
239	Inside Object Pascal	6/89
240	Using MPW for Non-Macintosh 68000 Systems	6/89
241	Script Manager’s Pixel2Char Routine	8/89
242	Fonts and the Script Manager	6/89
243	Script Manager Variables	6/89
244	A Leading Cause of Color Cursor Cursing	10/89
245	Font Family Numbers	3/91
246	Mixing HFS and C File I/O	8/89
247	Giving the (Desk)Hook to INITs	10/89
248	DAs & Drivers in Need of (a Good) Time	10/89
249	Opening the Serial Driver	12/89
250	AppleTalk Phase 2 on the Macintosh	12/89
251	Safe cdevs	8/89
252	Plotting Small Icons	10/89
253	'SICN' Tired of Large Icons in Menus?	10/89
254	Macintosh Portable PDS Development	2/90
255	Macintosh Portable ROM Expansion	10/89
256	Stand-Alone Code, *ad nauseam*	8/90
257	Slot Interrupt Prio-Technics	10/89
258	Our Checksum Bounced	10/89
259	Old Style Colors	8/90
260	NuBus Power Allocation	10/89
261	Cache As Cache Can	4/92
262	High-Level Control & Status Calls: When a Good Call Goes Bad	1/92
263	International Canceling	2/90
264	Script Manager 2.0 Date & Time Problems	2/90
265	Pascal to C: PROCEDURE Parameters	8/90
266	Absolute Pointing Device Memory Structure	4/92
267	TextEdit Technicalities	4/90
268	MacinTalk—The Final Chapter	2/90
269	'ckid' Resource Format	4/90

	270	AppleTalk Timers Explained	4/90
	271	Macintosh IIfx: The Inside Story	4/90
	272	What Your Sony Drives For You	6/90
	273	SCSI Termination	5/92
	274	The Compleat Guide to TeachText	4/90
	275	32-Bit QuickDraw: Version 1.2 Features	4/90
	276	Gimmie Depth or Gimmie Death	6/90
	277	Of Time and Space and _CopyBits	6/90
	278	_PBClose the Barn Door	6/90
	279	'LDEF' Madness	6/90
	280	"Bugs in MacApp? Yes, But I Love It!"	10/90
	281	Multiple Inheritance and HandleObjects	8/90
	282	Smear Tactics	8/90
	283	A/UX System Calls From Macintosh Software	1/91
	284	IOP-Based Serial Differences Under A/UX	8/90
	285	Coping With VM and Memory Mappings	4/91
	286	The Serial General-Purpose Input (GPi)	2/91
	287	Hey Buddy, Can You Spare A Block?	2/91
	288	NuBus Block Transfer Mode sResource Entries	2/91
	289	Deaccelerated _CopyBits & 8•24 GC QuickDraw	2/91
	290	Custom WDEF and wDraw	2/91
	291	CMOS On Macintosh LC PDS	2/91
	292	Bus Error Handlers	5/92
	293	Most Excellent CD Notes	2/91
	294	Me And My pIdle Proc (or how to let users know what's going on during print time...)	4/91
	295	Feeder Fodder	4/91
	296	The Lo Down on Dictionary Downloading	4/91
	297	Pictures and the Printing Manager	4/91
	298	Color, Windows and 7.0	5/92
	299	N/A	N/A
	300	My Life as a PascalObject	4/91
	301	File Sharing and Shared Folders	8/91
	302	Help for Movable Modal Dialogs	8/91
	303	Using a PurgeProc	8/91
	304	Pending Update Perils	10/91
	305	PBShare, PBUnshare, and PBGetUGEntry	1/92
	306	Drawing Icons the System 7 Way	5/92
	307	MPW C++ Pitfalls	5/92
	308	What Is a 'pslt' resource?	1/92
	309	Routes from the Source	1/92
	310	Who Put That Resource in My CDEV?	2/92
	311	What's New With AppleTalk Phase 2	4/92
	312	Fun with PrJobMerge	5/92
	313	Performance Tuning with Development Tools	5/92
	314	OmegaSANE	5/92
	315	Resolving Alias Files Quietly	5/92
	316	Data Access Extensions	5/92
***	317	FPU Operations on Macintosh Quadra Computers	6/92
***	318	Serial PollProc	6/92

ADB

143	Don’t Call ADBReInit on the SE with System 4.1	3/88
160	Key Mapping	2/91
206	Space Aliens Ate My Mouse (ADB–The Untold Story)	2/90
266	Absolute Pointing Device Memory Structure	4/92

Alias Manager

315	Resolving Alias Files Quietly	5/92

AppleShare

114	AppleShare and Old Finders	3/88
115	Application Configuration with Stationery Pads	3/88
116	AppleShare-able Apps. and the Resource Manager	3/88
137	AppleShare 1.1 Server FPMove Bug	3/88
165	Creating Files Inside an AppleShare Drop Folder	3/88
167	AppleShare Foreground Applications	3/88
216	AppleShare 1.1 and 2.0 Limits	10/88
305	PB Share, PBUnshare, and PBGetUGEntry	1/92

AppleTalk Manager

9	Will Your AppleTalk Application Support Internets?	4/90
20	Data Servers on Appletalk	3/88
121	Using the High-Level AppleTalk Routines	3/88
132	AppleTalk Interface Update	3/88
142	Avoid Use of Network Events	3/88
195	ASP and AFP Description Discrepancies	8/88
199	KillNBP Clarification	8/88
201	ReadPacket Clarification	8/88
224	Opening AppleTalk	2/89
225	Using RegisterName	2/89
250	AppleTalk Phase 2 on the Macintosh	12/89
270	AppleTalk Timers Explained	4/90
311	What’s New With AppleTalk Phase 2	4/92

Applications

84	Edit File Format	3/88
86	MacPaint Document Format	6/89
168	HyperCard And You: Economy Edition	2/91
274	The Compleat Guide to TeachText	4/90

A/UX

229	A/UX 2.0 Compatibility Guidelines	2/91
283	A/UX System Calls From Macintosh Software	1/91
284	IOP-Based Serial Differences Under A/UX	8/90

CD ROM

209	High Sierra & ISO 9660 CD ROM Formats	8/88
293	Most Excellent CD Notes	2/91

Compatibility

2	Compatibility Guidelines	3/88
25	Don't Depend on Register A5 Within Trap Patches	3/88
44	HFS Compatibility	3/88
83	System Heap Size Warning	3/88
100	Compatibility with Large-Screen Displays	3/88
103	Using MaxApplZone and MoveHHi from Assembly	3/88
117	Compatibility: Why and How	3/88
129	_Gestalt & _SysEnvirons—A Never-Ending Story	5/92
155	Handles and Pointers—Identity Crisis	3/88
156	Checking for Specific Functionality	3/88
194	WMgrPortability	4/88
203	Don't Abuse the Managers	8/88
208	Setting and Restoring A5	6/89
212	The Joy of Being 32-Bit Clean	8/91
213	_StripAddress: The Untold Story	8/90
227	Toolbox Karma	2/89
247	Giving the (Desk)Hook to INITs	10/89
248	DAs & Drivers in Need of (a Good) Time	10/89
249	Opening the Serial Driver	12/89

Control Manager

196	CDEF Parameters	10/89
310	Who Put That Resource in My CDEV?	2/92

Control Panel

215	"New" cdev Messages	10/88
251	Safe cdevs	8/89

Data Access Manager

316	Data Access Extensions	5/92

Debugging

7	A Few Quick Debugging Tips	3/88
38	The ROM Debugger	3/88
42	Pascal Routines Passed by Pointer	3/88

51	Debugging With PurgeMem and CompactMem	3/88
151	System Error 33, "zcbFree has gone negative"	3/88

Desk Accessories

5	Using Modeless Dialogs from Desk Accessories	3/88
23	Life With Font/DA Mover—Desk Accessories	3/88

Device Manager

36	Drive Queue Elements	3/88
56	Break/CTS Device Driver Event Structure	3/88
71	Finding Drivers in the Unit Table	3/88
187	Don't Look at ioPosOffset	4/88
197	Chooser Enhancements	8/88
257	Slot Interrupt Prio-Technics	10/89
262	High-Level Control & Status Calls: When a Good Call Goes Bad	1/92
272	What Your Sony Drives For You	6/90
278	_PBClose the Barn Door	6/90

Devices

*** 318	Serial PollProc	6/92

Dialog Manager

4	Error Returns from GetNewDialog	3/88
34	User Items in Dialogs	10/88
112	FindDItem	3/88
304	Pending Update Perils	10/91

Disk Initialization Package

70	Forcing Disks to be Either 400K or 800K	3/88
287	Hey Buddy, Can You Spare A Block?	2/91

Event Manager

3	Command-Shift-Number Keys	3/88
85	GetNextEvent; Blinking Apple Menu	3/88
202	Resetting the Event Mask	12/88
304	Pending Update Perils	10/91

File Manager

24	Available Volumes	3/88
66	Determining Which File System is Active	8/90
67	Finding the "Blessed Folder"	5/92
68	Searching Volumes-Solutions and Problems	1/92
69	Setting ioFDirIndex in PBGetCatInfo Calls	3/88
77	HFS Ruminations	3/88
81	Caching	3/88

87	Error in FCBPBRec	3/88
94	Tags	3/88
102	HFS Elucidations	3/88
106	The Real Story: VCBs and Drive Numbers	3/88
107	Nulls in Filenames	3/88
108	_AddDrive, _DrvrInstall, and _DrvrRemove	12/88
130	Clearing ioCompletion	3/88
140	Why PBHSetVol is Dangerous	3/88
157	Problem with GetVInfo	3/88
179	Setting ioNamePtr in File Manager Calls	3/88
186	Lock, Unlock the Range	2/91
190	Working Directories and MultiFinder	4/88
204	HFS Tidbits	8/88
218	New High-Level File Manager Calls	12/88
226	Moving Your Cat	2/91
238	Getting a Full Pathname	10/89
246	Mixing HFS and C File I/O	8/89
305	PB Share, PBUnshare, and PBGetUGEntry	1/92

Finder

306	Drawing Icons the System 7 Way	5/92

Font Manager

30	Font Height Tables	3/88
191	Font Names	8/88
198	Font/DA Mover, Styled Fonts, and NFNTs	5/92
245	Font Family Numbers	3/91

Hardware

10	Pinouts	3/88
65	Macintosh Plus Pinouts	3/88
144	Macintosh Color Monitor Connections	2/91
148	Suppliers for Macintosh II Board Developers	3/88
176	Macintosh Memory Configurations	4/92
221	NuBus Interrupt Latency (I was a teenage DMA junkie)	10/89
230	Pertinent Information About the Macintosh SE/30	6/89
234	NuBus Physical Designs—Beware	12/89
235	Cooperating with the Coprocessor	10/90
236	Speedy the Math Coprocessor	6/89
254	Macintosh Portable PDS Development	2/90
255	Macintosh Portable ROM Expansion	10/89
260	NuBus Power Allocation	10/89
261	Cache As Cache Can	4/92
271	Macintosh IIfx: The Inside Story	4/90
273	SCSI Termination	5/92
282	Smear Tactics	8/90
285	Coping With VM and Memory Mappings	4/91
286	The Serial General-Purpose Input (GPi)	2/91
288	NuBus Block Transfer Mode sResource Entries	2/91
291	CMOS On Macintosh LC PDS	2/91
292	Bus Error Handlers	5/92

International

List Manager

Memory Manager

Menu Manager

MPW & MacApp

MultiFinder

233	MultiFinder and _SetGrowZone	6/89

Notification Manager

184	Notification Manager	12/89

Palette Manager

211	Palette Manager Changes in System 6.0.2	10/88

Programming Languages & Tips

88	Signals	3/88
135	Getting through CUSToms	3/88
231	Macintosh Allegro Common Lisp Features	2/90
239	Inside Object Pascal	6/89
265	Pascal to C: PROCEDURE Parameters	8/90
300	My Life as a PascalObject	8/91

Printing Manager

33	ImageWriter II Paper Motion	3/88
72	Optimizing for the LaserWriter—Techniques	10/90
73	Color Printing	3/88
91	Optimizing for the LaserWriter—Picture Comments	3/88
92	The Appearance of Text	3/88
95	How to Add Items to the Print Dialogs	3/88
122	Device-Independent Printing	3/88
123	Bugs in LaserWriter ROMs	3/88
124	Using Low-Level Printing Calls With AT ImageWriters	3/88
125	Effect of Spool-a-page/Print-a-page on Shared Printers	3/88
128	PrGeneral	3/88
133	Am I Talking to a LaserShare Spooler?	3/88
149	Document Names and the Printing Manager	3/88
152	Using Laser Prep Routines	3/88
161	A Printing Loop That Cares…	10/90
173	PrGeneral Bug	3/88
175	SetLineWidth Revealed	3/88
183	Position-Independent PostScript	3/88
192	Surprises in LaserWriter 5.0 and Newer	2/90
217	Where Have My Font Icons Gone?	4/91
295	Feeder Fodder	4/91
297	Pictures and the Printing Manager	4/91
312	Fun with PrJobMerge	5/92

QuickDraw

21	QuickDraw's Internal Picture Definition	3/88
26	Fond of FONDS	5/92
41	Drawing Into an Off-Screen Bitmap	6/90
55	Drawing Icons	3/88
59	Pictures and Clip Regions	3/88
60	Drawing Characters in a Narrow GrafPort	3/88

	120	Principia Off-Screen Graphics Environment	3/92
	154	Displaying Large PICT Files	3/88
	163	Adding Color With CopyBits	3/88
	171	Things You Wanted to Know About _PackBits*	1/92
	181	Every Picture [Comment] Tells Its Story, Don't it?	3/88
	193	So Many Bitmaps, So Little Time	12/89
	244	A Leading Cause of Color Cursor Cursing	10/89
	259	Old Style Colors	8/90
	275	32-Bit QuickDraw: Version 1.2 Features	4/90
	276	Gimmie Depth or Gimmie Death	6/90
	277	Of Time and Space and _CopyBits	6/90
	289	Deaccelerated _CopyBits & 8•24 GC QuickDraw	2/91
	297	Pictures and the Printing Manager	4/91
	306	Drawing Icons the System 7 Way	5/92

Resource Manager

	6	Shortcut for Owned Resources	3/88
	32	Reserved Resource Types	3/88
	46	Separate Resource Files	3/88
	50	Calling SetResLoad	3/88
	62	Don't Use Resource Header Application Bytes	3/88
	74	Don't Use the Resource Fork for Data	3/88
	78	Resource Manager Tips	3/88
	101	CreateResFile and the Poor Man's Search Path	3/88
	141	Maximum Number of Resources in a File	3/88
	185	OpenRFPerm: What your mother never told you	4/88
	188	ChangedResource: Too much of a good thing	4/88
	214	New Resource Manager Calls	10/88
	232	Strip With _OpenResFile and _OpenRFPerm	4/89
	252	Plotting Small Icons	10/89
	253	'SICN' Tired of Large Icons in Menus?	10/89

SANE (Standard Apple Numerics Environment)

	314	OmegaSANE	5/92

SANE/Math Coprocessors

***	317	FPU Operations on Macintosh Quadra Computers	6/92

Script Manager

	174	Accessing the Script Manager Print Action Routine	3/88
	182	How to Construct Word-Break Tables	3/88
	241	Script Manager's Pixel2Char Routine	8/89
	242	Fonts and the Script Manager	6/89
	243	Script Manager Variables	6/89
	264	Script Manager 2.0 Date & Time Problems	2/90

SCSI Manager

	96	SCSI Bugs	3/88

159	Hard Disk Hacking	3/88
258	Our Checksum Bounced	10/89

Segment Loader

126	Sub(Launching) from a High-Level Language	4/89
220	Segment Loader Limitations	12/88

Sound Manager

19	How to Produce Continuous Sound Without Clicking	6/89
268	MacinTalk—The Final Chapter	2/90

Standard File Package

47	Customizing Standard File	3/88
80	Standard File Tips	3/88

System Software

28	Finders and Foreign Drives	3/88
29	Resources Contained in the Desktop File	3/88
40	Finder Flags	3/88
48	Bundles	3/88
75	Apple's Multidisk Installer	1/92
113	Boot Blocks	3/88
134	Hard Disk Medic & Booting Camp	3/88
139	Macintosh Plus ROM Versions	3/88
147	Finder Notes: "Get Info" Default & Icon Masks	3/88
189	Version Territory	10/90
210	The Desktop file's Outer Limits	8/88

TextEdit

18	TextEdit Conversion Utility	3/88
22	TEScroll Bug	3/88
82	TextEdit: Advice & Descent	3/88
127	TextEdit EOL Ambiguity	3/88
131	TextEdit Bugs in System 4.2	3/88
207	Styled TextEdit Changes in System 6.0	12/88
237	TextEdit Record Size Limitations Revisited	6/89
267	TextEdit Technicalities	4/90

Token Ring

309	Routes from the Source	1/92

Window Manager

79	_ZoomWindow	4/90
290	Custom WDEF and wDraw	2/91
298	Color, Windows and 7.0	5/92

Macintosh
Technical Notes

Developer Technical Support

#250: AppleTalk Phase 2 on the Macintosh

Revised by: Sriram Subramanian December 1989
Written by: Pete Helme & Sriram Subramanian August 1989

This Technical Note discusses the new features and calls available with AppleTalk Phase 2.
Changes since August 1989: Incorporated the `ClosePrep` and `CancelClosePrep` transitions and the new control calls to the .MPP driver.

AppleTalk Phase 2 is only available on Macintosh Plus or later Macintosh platforms, and it requires the installation of AppleTalk file V53, or greater. Both EtherTalk 2.0 and TokenTalk 2.0 automatically install this AppleTalk file. Developer Technical Support can supply the Phase 2 drivers for development use; however, if you need to include the Phase 2 drivers in your product, you must license them from Software Licensing. For more information, contact:

Apple Software Licensing
Apple Computer, Inc.,
20525 Mariani Avenue, M/S 38-I
Cupertino, CA, 95014
(408) 974-4667
AppleLink: SW.LICENSE

What is AppleTalk Phase 2?

AppleTalk Phase 2 contains enhancements to the routing and naming services of AppleTalk. Among these enhancements is the ability to create AppleTalk networks which support more than 254 nodes, and to do so in a manner that is, to the greatest extent possible, compatible with current AppleTalk implementations and applications. Multiple zones per network are now supported, and users can choose their machine's zone. Benefits include improved network traffic and better router selection. New calls and features have been implemented with this enhancement and are documented in this Note. The *AppleTalk Phase 2 Protocol Specification*, which details the changes to the AppleTalk protocol suite, is available from Developer Technical Support at the address listed in Technical Note #0.

Are AppleTalk Phase 2 Drivers Present?

So you want to use these new calls and features, but can you? First, one needs to check to see if the node is running AppleTalk Phase 2. There are two ways this can be accomplished. The easiest way is to make a `_SysEnvirons` call and check the returned `atDrvrVersNum` field. If this byte is greater than or equal to 53, then AppleTalk Phase 2 drivers are present. If, for some reason, a `_SysEnvirons` call is not practical or otherwise not possible, one can check 7 bytes off the device control entry for the .MPP driver for a single byte, which is the driver version (actually the low byte of the qFlags field of DCtlQHdr in the DCE). Again, if this byte is 53 or greater, AppleTalk Phase 2 is present, and the calls and features outlined in this Note may be used.

Calls to the .MPP Driver

AppleTalk Phase 2 introduces many new variables, and we highly recommend that you use the new GetAppleTalkInfo call instead of looking at MPP globals directly. In addition, on a Macintosh running the AppleTalk Internet Router software, there may be more than one .MPP driver present. These additional drivers can be found by walking through the unit table (UTableBase $11C) and looking for drivers named .MPP other than at unit slot 9. Generally, the only port of interest to you is the user port, reflected in this call as PortID 0 with a refnum of -10.

GetAppleTalkInfo

```
Parameter Block
    --> 26     csCode           word       ; always GetAppleTalkInfo (258)
    --> 28     Version          word       ; requested info version
    <-- 30     VarsPtr          pointer    ; pointer to well known MPP vars
    <-- 34     DCEPtr           pointer    ; pointer to MPP DCE
    <-- 38     PortID           word       ; port number [0..7]
    <-- 40     Configuration    long       ; 32-bit configuration word
    <-- 44     SelfSend         word       ; non zero if SelfSend enabled
    <-- 46     NetLo            word       ; low value of network range
    <-- 48     NetHi            word       ; high value of network range
    <-- 50     OurAddr          long       ; our 24-bit AppleTalk address
    <-- 54     RouterAddr       long       ; 24-bit address of (last) router
    <-- 58     NumOfPHs         word       ; max. number of protocol handlers
    <-- 60     NumOfSkts        word       ; max. number of static sockets
    <-- 62     NumNBPEs         word       ; max. concurrent NBP requests
    <-- 64     NTQueue          pointer    ; pointer to registered name queue
    <-> 68     *LAlength        word       ; length in bytes of data link addr
    --> 70     *LinkAddr        pointer    ; data link address returned
    --> 74     *ZoneName        pointer    ; zone name returned
               * for extended networks only
```

This call is provided to simplify the task of obtaining details about the current AppleTalk network connection. The following are the parameters which this call returns:

Version — is passed by the caller. The concept is similar to one used by _SysEnvirons, where a version ID is passed to the function to return a requested level of information. If the driver cannot respond because this number is too high, paramErr is returned. The current version number is 1.

VarsPtr — is the pointer to AppleTalk variables. This points to the well known sysLapAddr and read header area or RHA.. This pointer may not be equal to $2D8 (ABusVars) for other than port 0.

DCEPtr — is a pointer to the driver's device control entry. See the Device Manager chapters of *Inside Macintosh* for details.

PortID — is the port number, and it is always zero, unless a router is active and a driver refnum other than -10 is used.

Configuration — is a 32-bit word of configuration flags. Currently only the following bits are returned:

31 (SrvAdrBit) — is true if server node-ID was requested at open time. Note that even if server address is requested, it may be ignored

by those ADEVs which do not honor it (i.e., EtherTalk, TokenTalk, etc.).

30 (`RouterBit`) is true if an AppleTalk Internet Router was loaded at system startup. Note that a router may be loaded, but not active.

7 (`BadZoneHintBit`) is true if the node's zone name hint is invalid, thus causing a default zone to be selected.

6 (`OneZoneBit`) is true if only one zone is assigned to an extended network.

`SelfSend` (the ability for a node to send packets to itself) is non-zero if this feature is currently enabled.

`NetLo` is the low value of the network range. Non-extended networks always have a range of exactly one network, if the network number is known.

`NetHi` is the high value of the network range.

`OurAddr` is the 24-bit AppleTalk network address of the node. The most significant byte is always zero.

`RouterAddr` is the 24-bit AppleTalk address of the router from which we last heard. Users should always use this address when attempting to communicate directly with a router.

`NumOfPHs`, `NumOfSkts`, and `NumNBPEs` are maximum capacities for the driver. They are number of protocol handlers, number of static sockets, and number of concurrent NBP requests allowed, respectively.

`NTQueue` is a pointer to the registered names table queue. See *Inside Macintosh*, Volume II, The AppleTalk Manager, for NT Queue details.

`LALength` is passed by the caller to indicate how much (if any) of the data link address is to be copied to a user-suppled buffer (pointed to by `LinkAddr`). The actual length is returned by the driver. If the caller requests more bytes than the actual number, then data in the buffer after the address is undefined. The caller is responsible for providing sufficient buffer space.

`LinkAddr` is a pointer to a user-supplied buffer into which the data link address data is copied. If the pointer is `NIL`, no data is copied.

`ZoneName` is a pointer to a user-supplied buffer into which the node's stored zone name is copied. If the pointer is `NIL`, no data is copied. The user buffer must be 33 bytes or more in size.

Calls to the .ATP Driver

KillAllGetReq

```
Parameter Block
    --> 26     csCode          word       ; always KillAllGetReq (259)
    --> 28     atpSocket       byte       ; socket on which to kill all pending GetRequests
```

`KillAllGetReq` aborts all outstanding `GetRequest` calls on the specified socket and completes them with `reqAborted` errors (it does not close the specified socket, it only kills all pending `GetRequest` calls on that socket). To kill all the `GetRequest` calls, simply pass the desired socket number in the `atpSocket` field.

Result codes	noErr	No Error	(0)
	cbNotFound	control block not found	(-1102)

Setting the TRel Timer in SendRequest Calls

It is now possible to set the TRel timer in SendRequest or NSendRequest calls with ATP XO (exactly once) service so as not to be locked into the pre-AppleTalk Phase 2 time of 30 seconds. This is done by setting bit 2 in the atpFlags field to indicate to the driver that an extended parameter block is being used. Make a standard SendRequest call, but add the timeout constant desired in the new TRelTime field byte of the parameter block. Both nodes must be running AppleTalk Phase 2 for this feature to be supported.

The timeout constants are enumerated as follows in the lower three bits of the TRelTime ($32 offset) byte:

000	$0	TRel timer set to 30 seconds
001	$1	TRel timer set to one minute
010	$2	TRel timer set to two minutes
011	$3	TRel timer set to four minutes
100	$4	TRel timer set to eight minutes

All other values are reserved.

```
Parameter Block
   --> 50     TRelTime          byte       ; indicates time to wait for TRel packet
```

Name Binding Protocol (NBP) Change: Wildcard Lookup

In AppleTalk Phase 2, NBP is enhanced to provide additional wildcard support. The double tilde (≈), $C5, is now reserved in the object name and type strings and used in a lookup to mean a match of zero or more characters. Thus "≈cliff" matches "cliff," "the cliff," "grazing off the cliff," etc., and "123≈456" matches "123456," "123zz456," etc. At most one ≈ is allowed in any string. A single ≈ has the same meaning as a single =, which also must continue to be accepted. The ≈ has no special meaning in zone names. Clients of NBP must be aware that "old" (pre-AppleTalk Phase 2) nodes may not process this new wildcard feature correctly. This feature should probably only be used when it is known that the responding devices are running Phase 2 drivers as well.

Obtaining Zone Information Using the New .XPP Driver Calls

Previously, Zone Information Protocol (ZIP) functions were accomplished via direct ATP calls to the local router. It was rather nasty business, having to mess with the ATPUserData on subsequent calls to retain state information. We now recommend the use of the following XPP driver calls to access ZIP. Old ATP calls will continue to be supported for compatibility. It should also be noted that with Phase 2 drivers present, the .XPP driver is automatically opened by MPP.

GetZoneList

```
Parameter Block
    --> 26     csCode          word       ; always xCall (246)
    --> 28     xppSubCode      word       ; always zipGetZoneList (6)
    --> 30     xppTimeout      byte       ; retry interval (seconds)
    --> 31     xppRetry        byte       ; retry count
        32     <unused>        word       ; word space for rent.  see the super.
    --> 34     zipBuffPtr      pointer    ; pointer to buffer (must be 578 bytes)
    <-- 38     zipNumZones     word       ; no. of zone names in this response
    <-- 40     zipLastFlag     byte       ; non-zero if no more zones
        41     <unused>        byte       ; filler
    --> 42     ziplnfoField    70 bytes   ; on initial call, set first word to zero
```

`GetZoneList` is used to obtain a complete list of zones on the internet. `ZipBuffPtr` points to a buffer that.must be 578 bytes (`ATPMaxData`) in length. The actual number of zone names returned in the buffer is returned in `zipNumZones`. The fields `xppTimeout` and `xppRetry` contain the ATP retry interval (in seconds) and count, respectively.

The first time this call is made, the first word of the `ziplnfoField` should be set to zero. When the call completes, `zipLastFlag` is non-zero if all the zone names fit into the buffer. If not, the call should be made again immediately, without changing `zipInfoField` (it contains state information needed to get the next part of the list). The call should be repeated until `zipLastFlag` is non-zero. The 70-byte `zipInfoField` must always be allocated at the end of the parameter block.

Result codes	`noErr`	No Error	(0)
	`noBridgeErr`	No router is available	(-93)
	`ReqFailed`	SendRequest failed; retry count exceeded	(-1096)

Following are short examples of using `GetZoneList`.

Pascal

```
 const
{ csCodes for new .XPP driver calls }
  xCall = 246;

{ xppSubCodes }
  zipGetLocalZones = 5;
  zipGetZoneList = 6;
  zipGetMyZone = 7;

 type
{ offsets for xCall queue elements  }
   xCallParam = packed record
    qLink: QElemPtr;
    qType: INTEGER;
    ioTrap: INTEGER;
    ioCmdAddr: Ptr;
    ioCompletion: ProcPtr;
    ioResult: OsErr;
    ioNamePtr: StringPtr;
    ioVRefNum: INTEGER;
    ioRefNum: INTEGER;
    csCode: INTEGER;
    xppSubCode: INTEGER;
    xppTimeOut: Byte;
    xppRetry: Byte;
    filler: INTEGER;
    zipBuffPtr: Ptr;
    zipNumZones: INTEGER;
    zipLastFlag: INTEGER;
    zipInfoField: packed array[1..70] of Byte;
   end;

procedure doGetZoneListPhs2;

type
   XCallParamPtr = ^XCallParam;
var
   xpb: XCallParamPtr;
   resultCode: OSErr;
   zoneBuffer, theBufferPtr: Ptr;
   totalZones: integer;
begin
   xpb := XCallParamPtr(NewPtr(sizeof(XCallParam)));

   zoneBuffer := NewPtr(33 * 100);  { size of maxstring * 100 zones }

   theBufferPtr := NewPtr(578);     { size of atpMaxData }

   xpb^.zipInfoField[1] := 0;       { ALWAYS 0 on first call.  contains state info on}
                                    { subsequent calls }

   xpb^.zipInfoField[2] := 0;       { ALWAYS 0 on first call.  contains state info on }
                                    {subsequent calls }

   xpb^.ioRefNum := XPPRefNum;             { driver refNum -41 }

   xpb^.csCode := xCall;
   xpb^.xppSubCode := zipGetZoneList;
   xpb^.xppTimeOut := 3;
   xpb^.xppRetry := 4;
   xpb^.zipBuffPtr := Ptr(theBufferPtr);   { this buffer will be filled with }
                                           { packed zone names }

{ initialization for loop }
   xpb^.zipLastFlag := 0;
   totalZones := 0;
   resultCode := 0;

{ loop until zipLastFlag is non-zero or an error occurs }
   while ((xpb^.zipLastFlag = 0) and (resultCode = 0)) do
   begin
   resultCode := PBControl(ParmBlkPtr(xpb), false);

   if (resultCode = noErr) then
     begin
       totalZones := xpb^.zipNumZones + totalZones;
       { you can now copy the zone names into the zoneBuffer }
     end;
   end;
   DisposPtr(theBufferPtr);
   DisposPtr(zoneBuffer);
   DisposPtr(Ptr(xpb));
end;
```

C

```
/*
csCodes for new .XPP driver calls
*/
#define xCall               246

/*
xppSubCodes
*/
#define zipGetLocalZones    5
#define zipGetZoneList      6
#define zipGetMyZone        7

/*
offsets for xCall queue elements
*/
typedef struct
    {
    QElemPtr                qLink;
    short                   qType;
    short                   ioTrap;
    Ptr                     ioCmdAddr;
    ProcPtr                 ioCompletion;
    OsErr                   ioResult;
    StringPtr               ioNamePtr;
    short                   ioVRefNum;
    short                   ioRefNum;
    short                   csCode;
    short                   xppSubCode;
    unsigned char           xppTimeOut;
    unsigned char           xppRetry;
    short                   filler;
    Ptr                     zipBuffPtr;
    short                   zipNumZones;
    short                   zipLastFlag;
    unsigned char           zipInfoField[70];
} xCallParam;

doGetZoneListPhs2()
{
    xCallParam              xpb;
    OSErr                   resultCode = 0;
    Ptr                     zoneBuffer, theBufferPtr;
    short                   totalZones = 0;

    zoneBuffer = NewPtr(33*100);          /* size of maxstring * 100 zones */

    theBufferPtr = NewPtr(578);           /* size of atpMaxData */

    xpb.zipInfoField[0] = 0;              /* ALWAYS 0 on first call.  contains state info
                                          on subsequent calls */
    xpb.zipInfoField[1] = 0;              /* ALWAYS 0 on first call.  contains state info
                                          on subsequent calls */

    /* initialization for loop */
    xpb.zipLastFlag = 0;

    xpb.ioCRefNum = XPPRefNum;            /* driver refNum -41 */
    xpb.csCode = xCall;
    xpb.xppSubCode = zipGetZoneList;
    xpb.xppTimeOut = 3;
    xpb.xppRetry = 4;
    xpb.zipBuffPtr = (Ptr) theBufferPtr;  /* this buffer will be filled with
                                          the packed zone names */
```

```
        /* loop until zipLastFlag is non-zero or an error occurs */
        while(xpb.zipLastFlag == 0 && resultCode == 0) {

            resultCode = PBControl(&xpb, false);

            if(resultCode == noErr) {
                   totalZones += xpb.zipNumZones;
                   /* you can now copy the zone names into the zoneBuffer */
               }
    DisposPtr(theBufferPtr);
    DisposPtr(zoneBuffer);
    }
}
```

GetLocalZones

```
Parameter Block
    --> 26     csCode           word        ; always xCall (246)
    --> 28     xppSubCode       word        ; always zipGetLocalZones (5)
    --> 30     xppTimeout       byte        ; retry interval (seconds)
    --> 31     xppRetry         byte        ; retry count
        32     <unused>         word        ; filler
    --> 34     zipBuffPtr       pointer     ; pointer to buffer (must be 578 bytes)
    <-- 38     zipNumZones      word        ; no. of zone names in this response
    <-- 40     zipLastFlag      byte        ; non-zero if no more zones
        41     <unused>         byte        ; filler
    --> 42     zipInfoField     70 bytes    ; on initial call, set first word to zero
                                            ; on subsequent calls, do not modify!
```

This call has the same format and procedures as `GetZoneList`, the difference being that `GetLocalZones` returns a list of zone names currently defined only on the node's network cable rather than the entire network. The 70-byte `zipInfoField` must always be allocated at the end of the parameter block.

Result codes	noErr	No Error	(0)
	noBridgeErr	No router is available	(-93)
	ReqFailed	SendRequest failed; retry count exceeded	(-1096)

Note: The examples for `GetZoneList` will also work for `GetLocalZones` if you substitute the `xppSubCode`.

GetMyZone

```
Parameter Block
    --> 26     csCode           word        ; always xCall (246)
    --> 28     xppSubCode       word        ; always zipGetMyZone (7)
    --> 34     zipBuffPtr       pointer     ; pointer to buffer (must be 33 bytes)
    --> 42     zipInfoField     70 bytes    ; first word must be set to zero on every call
```

`GetMyZone` returns the node's AppleTalk zone name. This is the zone in which all of the node's network visible entities are registered. `ZipBuffPtr` points to a buffer that must be 33 bytes in length. If `noBridgeErr` is returned by the call, there is no internet, and the zone name is effectively an asterisk (*). The 70-byte `zipInfoField` must always be allocated at the end of the parameter block.

Result codes	noErr	No Error	(0)
	noBridgeErr	No router is available	(-93)
	ReqFailed	SendRequest failed; retry count exceeded	(-1096)

Following are short examples of using `GetMyZone`.

Pascal

```
procedure getMyZonePhs2;
var
   xpb:xCallParam;
   resultCode :OSErr;
   myZoneNameBuffer:Ptr;
begin
   myZoneNameBuffer  := NewPtr(33);

   xpb.ioCRefNum := xppRefNum;
   xpb.csCode := xCall;
   xpb.xppSubCode := zipGetMyZone;
   xpb.zipBuffPtr := myZoneNameBuffer;
   xpb.zipInfoField[1] := 0;                { ALWAYS 0 }
   xpb.zipInfoField[2] := 0;                { ALWAYS 0 }
   resultCode := PBControl(@xpb, false);
end;
```

C

```
getMyZonePhs2()
{
    xCallParam        xpb;
    OSErr             resultCode;
    Ptr               myZoneNameBuffer;

    myZoneNameBuffer  := NewPtr(33);

    xpb.ioCRefNum = xppRefNum;
    xpb.csCode = xCall;
    xpb.xppSubCode = zipGetMyZone;
    xpb.zipBuffPtr = (Ptr) myZoneNameBuffer;
    xpb.zipInfoField[0] = 0;                /* ALWAYS 0 */
    xpb.zipInfoField[1] = 0;                /* ALWAYS 0 */
    resultCode = PBControl(&xpb, false);
}
```

Potential Nastiness

When running on a node with Phase 2 compatible drivers, we always recommend using the .XPP calls outlined in the previous section. Care was taken to keep backward compatibility with the already existing ATP ZIP calls (they are being trapped out with the Phase 2 drivers), but there are problems about which you should be aware.

- Do not rely on checking the TID (transaction ID validity bit) or other bits in the `atpFlags`, as some of you have been doing. The `atpFlags` are not guaranteed to be correct on an ATP ZIP call with a Phase 2 driver present.
- Do not repeatedly stuff the router address back into the `ATPParamBlock` on subsequent ATP ZIP `GetZoneList` calls. There exists the possibility of concurrent `GetZoneList` calls being made by other tasks and wrong router addresses being used (a small possibility yes, but it does exist).

The AppleTalk Transition Queue

To keep applications and other resident processes on the Macintosh informed of AppleTalk events, such as the opening and closing of AppleTalk drivers, a new transition queue has been implemented. Processes can register themselves with the AppleTalk Transition Queue, and when a significant event occurs, they will be notified of this fact. Each transition queue element has the following MPW assembly-language format:

```
AeQentry       RECORD       0
QLink          DS.L         1       ; link to next record
QType          DS.W         1       ; unused
CallAddr       DS.L         1       ; pointer to task record
               ENDR
```

Three calls have been provided in the LAP Manager to add an entry, remove an entry, and return a pointer to the AppleTalk event queue header. The method for making calls to the LAP Manager is explained in the following section. The queue is maintained by the LAP Manager, so it can be active even when AppleTalk (MPP) is not.

Making a LAP Manager Call

The LAP Manager is installed in the system heap at startup time, before the AppleTalk Manager opens the .MPP driver (hence, the inclusion of the AppleTalk Transition Queue in LAP Manager rather than under .MPP). Calls are made to the LAP Manager by jumping through a low-memory location, with register `D0` equal to a dispatch code that identifies the function. The exact sequence is:

```
               MOVEQ   #Code,D0            ; D0 = ID code of wanted LAP call
               MOVE.L  LAPMgrPtr,An        ; An -> start of LAP manager (from $B18)
               JSR     LAPMgrCall(An)      ; Call the LAP manager at entry point

LAPMgrPtr      EQU     $B18                ; This points to our start (more
                                           ; commonly known as ATalkHk2)
LAPMgrCall     EQU     2                   ; Offset to make LAP manager calls
```

The AppleTalk Transition Queue LAP Calls

LAddAEQ (`D0`=23)
Call: `A0-->`   Entry to be added to the AppleTalk event queue.

The `LAddAEQ` call adds an entry, pointed to by `A0`, to the AppleTalk event queue.

```
               MOVEQ   #LAddAEQ,D0         ; D0 = 23 code of LAddAEQ LAP call
               MOVE.L  LAPMgrPtr,An        ; An -> start of LAP manager (from $B18)
               JSR     LAPMgrCall(An)      ; Call the LAP manager at entry point
```

LRmvAEQ (`D0`=24)
Call: `A0-->`   Entry to be removed from the AppleTalk event queue.

The `LRmvAEQ` call removes an entry, pointed to by `A0`, from the AppleTalk event queue.

```
               MOVEQ   #LRmvAEQ,D0         ; D0 = 24 code of LRmvAEQ LAP call
               MOVE.L  LAPMgrPtr,An        ; An -> start of LAP manager (from $B18)
               JSR     LAPMgrCall(An)      ; Call the LAP manager at entry point
```

LGetAEQ (D0=25)
Return: A1--> Pointer to the AppleTalk event queue header.

The LGetAEQ call returns a pointer in A1 to the AppleTalk event queue header, previously described.

```
MOVEQ   #LGetAEQ,D0         ; D0 = 25 code of LGetAEQ LAP call
MOVE.L  LAPMgrPtr,An        ; An -> start of LAP manager (from $B18)
JSR     LAPMgrCall(An)      ; Call the LAP manager at entry point
```

The Transitions

Each process is called at CallAddr when any significant transitions occur. A value is passed in, which indicates the nature of the event. Additional parameters may also be passed and a pointer to the task's queue element is also passed. This is provided so processes may append their own data structures (e.g., a globals pointer) at the end of the task record, which can be referenced when they are called. Processes should follow the MPW C register conventions. Registers D0, D1, D2, A0, and A1 are scratch registers that are not preserved by C functions. The arguments passed to the process should be left on the stack, since the calling routine removes them. All other registers should be preserved.

The Open Transition

For AppleTalk open transitions, the process has the following interface:

From assembly language, the stack upon calling looks as follows:

```
OpenEvent     RECORD      0
ReturnAddr    DS.L        1       ; address of caller
theEvent      DS.L        1       ; = 0 ; ID of Open transaction
aqe           DS.L        1       ; pointer to task record
SlotDevParam  DS.L        1       ; pointer to Open parameter block
              ENDR
```

This routine is called **only** when the open routine for .MPP executes successfully. Every entry in the transition queue is called in the same order that the entries were added to the queue. If AppleTalk is already open and an _Open call is made, no process is called. The process should return a function result in D0, which is currently ignored.

A pointer to the open request parameter block is passed to the open event process for information only (i.e., the event process may not prevent AppleTalk open calls). Those fields which are of interest are OpenPB->ioPermssn, passed by the caller, and OpenPB->ioMix, which is both passed by the caller and updated by the .MPP open (see *Inside Macintosh*, Volume V, The AppleTalk Manager).

The Close Transition

For AppleTalk close transitions, the process has the following interface:

From assembly language, the stack upon calling looks as follows:

```
CloseEvent    RECORD      0
ReturnAddr    DS.L        1       ; address of caller
theEvent      DS.L        1       ; = 2 ; ID of Close transaction
aqe           DS.L        1       ; pointer to task record
              ENDR
```

The process is being told that AppleTalk is closing, which gives the process an opportunity to close gracefully. Every entry in the event queue is called, one after the other, in the same order that the entries were added to the queue. The close action cannot be cancelled. The process should return a function result in `D0`, which is currently ignored.

The ClosePrep and CancelClosePrep Transitions

The `AtalkClosePrep` and the `CancelAtalkClosePrep` control calls are used by various elements of the System, such as the Chooser, to inform or query AppleTalk clients of the closing of network drivers. For example, on a machine equipped to go to sleep or to wake up, the `_Sleep` trap is used by such entities as sleeptimer, Finder, and Shutdown to inform AppleTalk clients that it is desirable for the the network driver (.MPP) to be closed. The `_Sleep` trap may be trying to do any of the following three things: request permission for sleep, alert for impending sleep, or inform that wake up is underway. The sleep request calls the following two .MPP control calls; these calls are made before sleep queue procedures are called.

The first control call, `AtalkClosePrep`, is used to inform or query AppleTalk clients that the network driver might be closed in the very near future. The call has the following interface:

AtalkClosePrep (`csCode = 259`)

```
Parameter Block
    --> 26      csCode          word        ;always AtalkClosePrep
    <-- 28      clientName      pointer     ;-> name of client using driver
```

Result codes	`noErr`	The AppleTalk network driver (.MPP) may be closed
	`closeErr`	The AppleTalk network driver (.MPP) may not be closed

`clientName` is a pointer to an identifying string that is returned only if the result is `closeErr`. Note that the pointer may be `NIL` in this case, while the pointer is always `NIL` if the return code is `noErr`.

All tasks in the AppleTalk Transition Queue are called with the event `ClosePrep`. The tasks can prevent driver closure with a negative response to the event call. Each task is called with the following interface:

From assembly language, the stack upon calling looks as follows:

```
ClosePrep           RECORD      0       ;top of the stack
ReturnAddr          DS.L        1       ;addr of caller
theEvent            DS.L        1       ;=3
aqe                 DS.L        1       ;->task rec.
clientName          DS.L        1       ;ptr. to ptr. to name of client
                    ENDR
```

For this event, `theEvent = 3`, and the task is being **both** informed and asked if closing the network driver is acceptable. If driver closure is acceptable, the task need only to reply affirmative (`D0 = 0`), or if not acceptable, deny the request (`D0 ≠ 0`). The task may use the event as an opportunity to "prepare to die" or may simply respond. For example, a task may prevent further sessions from forming while waiting for the actual close event.

`clientName` is a pointer to a field in the .MPP control call parameter block where the task may optionally store a string address. This string identifies the client who has AppleTalk in use and is denying the request to close it. This string may be used in a dialog to inform the user to take appropriate action or explain why the requested action could not be performed.

If any task responds negatively, no subsequent tasks are called. Any tasks called prior to the one that denied a query are recalled with another event, `CancelClosePrep` (described below), enabling them to "undo preparations to die," and the control call then completes with a `closeErr` error.

From assembly language, the stack upon calling looks as follows:

```
CancelClosePrep     RECORD      0       ;top of the stack
ReturnAddr          DS.L        1       ;addr of caller
theEvent            DS.L        1       ;=4
aqe                 DS.L        1       ;->task rec.
                    ENDR
```

For this event, `theEvent = 4`, and the task is being informed that although it has recently approved a request to close the network driver, a subsequent task in the AppleTalk Transition Queue has denied permission. This event permits the task to undo any processing that may have been performed in anticipation of the network driver being closed. The process should return a function result in `D0`, which is currently ignored.

The second new control call, `CancelAtalkClosePrep`, is used to undo the effects of a successful `AtalkClosePrep` control call. Even though all queried tasks in the AppleTalk Transition Queue approved of network driver closure, other conditions may exist after making the `AtalkClosePrep` control call which prohibit network driver closure. In this case, it is necessary to recall all tasks to undo any processing that may have been performed in anticipation of the network driver being closed. The control call to do this has the following interface:

CancelAtalkClosePrep `(csCode = 260)`

```
Parameter Block
    --> 26      csCode              word        ;always CancelAtalkClosePrep
```

Result codes `noErr` Nothing could possibly go wrong

All tasks in the AppleTalk Transition Queue are called with the event `CancelClosePrep` as described above.

Note: The use of the low-memory global `ChooserBits` ($946) is no longer an acceptable means of preventing AppleTalk from closing when AppleTalk Phase 2 is present. Transitions other than defined above must be ignored and are reserved for future implementation. In the future transitions may be defined for notifying processes when a change in zone name occurs.

Potential Compatibility Problems

Using DDP and Talking to Routers

If, for some reason, you need to talk to any router via DDP, always use the `GetAppleTalkInfo` call outlined in this Note to get the router's actual 24-bit address.

The `WriteLAP` function (`csCode` = 243) to the .MPP driver is no longer supported, since a node is no longer identified only by its eight-bit (LAP) node ID.

On a Macintosh running the AppleTalk Internet Router software, the `SelfSend` flag is always set, so if you try to clear this flag using the `PSetSelfSend` call (Inside Macintosh, Volume V-514), you will get an error.

Further Reference:

- *Inside AppleTalk*
- *Inside Macintosh*, Volume II, The AppleTalk Manager
- *Inside Macintosh*, Volume V, The AppleTalk Manager
- *EtherTalk and Alternate AppleTalk Connections Reference*, May 5, 1989—Draft (DTS)
- AppleTalk Phase 2 Protocol Specification (DTS)
- Macintosh Portable Developer Notes (DTS)

Macintosh
Technical Notes

Developer Technical Support

#251: Safe cdevs

Written by: John Harvey

August 1989

This Technical Note describes a potential problem with Control Panel devices (cdevs) that contain `EditText` fields and presents a way to avoid it.

The Control Panel chapter in *Inside Macintosh*, Volume 5 describes, in detail, how run-time errors are handled by the Control Panel and a cdev. There is, however, a potential problem with cdevs that contain `EditText` items that this chapter does not cover.

When a cdev is called by the Control Panel, the cdev's `'DITL'` resource is concatenated to the Control Panel's `'DITL'`. The Control Panel then lets the Dialog Manager update the window. If the cdev contains an item of type `EditText`, the Dialog Manager allocates and activates a `TEHandle` to be used for displaying and editing text. All of this action happens before the cdev gets the `initDev` message from the Control Panel.

As detailed in The Control Panel chapter, if an error occurs from which a cdev cannot recover, the cdev should dispose of any private memory and return the appropriate error code or a `NIL` value to the Control Panel. The Control Panel then grays out the cdev's area, displays the appropriate error dialog, and then deletes the items that were added to its `'DITL'`.

All of this is fine, except that the `TEHandle` does not get deallocated. The `EditText` items get thrown away, including the strings in the item list that the Dialog Manager would use to store text entered into the `EditText` field, but the `TEHandle` stays there and stays active. Figure 1 illustrates what this would look like.

This line is a result of _DialogSelect calling _TEIdle

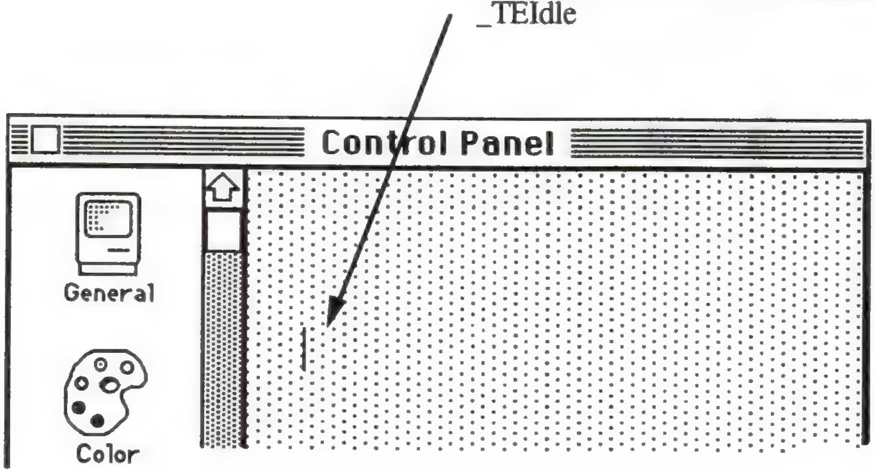


Figure 1–Erroneous Insertion Point

So the Dialog Manager, knowing that it allocated a `TEHandle` for an item that was visible, goes merrily on its way flashing the insertion point. The problem is not simply one of appearance. If a user hits a key, the Dialog Manager tries to process the key-down event just as if the `EditText` item was still available, and this series of events causes a rather nasty crash.

Fortunately, the solution for this problem is a very simple one. If an `EditText` item is hidden with a `_HideDItem` call, the Dialog Manager does not consider it active and will not try to process key-down events for it. So if your cdev contains `EditText` items, part of your error handling should be to first hide the `EditText` items with a call to `_HideDItem` before returning an error code or a `NIL` as the cdev's function result.

Further Reference:

- *Inside Macintosh*, Volume I, The Dialog Manager
- *Inside Macintosh*, Volume IV, The Dialog Manager
- *Inside Macintosh*, Volume V, The Control Panel

Macintosh
Technical Notes

Developer Technical Support

#252: Plotting Small Icons

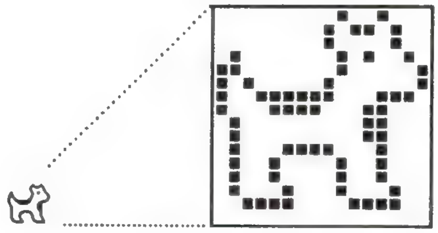
Revised by: James Beninghaus — October 1989
Written by: James Beninghaus & Dennis Hescox — August 1989

This Technical Note discusses the `'SICN'` resource format and how to plot one in a `GrafPort`.
Changes since August 1989: Corrected errors in the Pascal code and spruced up the rest.

Introduction

Apple first introduced the `'SICN'` resource so that the Script Manager could represent which country specific resources are installed in the system by displaying a small icon in the upper right corner of the menu bar. You can pass a `'SICN'` resource to the Notification Manager or Menu Manager, and they will draw it for you automatically—you should continue to let them do so. However, if you want to draw a small icon in your application's window, then this Note can help.

What does a `'SICN'` look like? Following is a `'SICN'` representation of a dogcow to help answer this question:



`'SICN'` FatBits

There is reason to believe that this representation is actually a baby dogcow. Due to the protective nature of parent dogcows, young dogcows are rarely seen. This one was spotted during a DTS meeting after it drew attention to itself by crying “moo! woof!”. (Note that this dogcow said “moo! woof!” because it was immature; adult dogcows naturally say, “Moof!”.)

'SICN' Resource

A 'SICN' resource contains any number of small icon bit images. Each small icon in a 'SICN' list describes a 16 by 16 pixel image and requires 32 bytes of storage. Like an 'ICN#' resource, there is no count of the number of icons stored in a 'SICN'. The following 'SICN' resource, in MPW Rez format, contains two small icons:

```
resource 'SICN' (1984, "clarus") {
        {       /* array: 2 elements */

                $"00 48 00 B4 00 84 40 52 C0 41 A0 81 9F 8E 8F 18"
                $"40 18 40 18 47 88 48 48 48 48 44 44 3C 3C 00 00",

                $"00 48 00 FC 00 FC 40 7E C0 7F E0 FF FF FE FF F8"
                $"7F F8 7F F8 7F F8 78 78 78 78 7C 7C 3C 3C 00 00"
        }
};
```

The Right Tools for the Job

The Macintosh Toolbox interfaces do not describe all the necessary data structures needed to work with 'SICN' resources. As shown in the following example, defining the 'SICN' type as an array of 16 short integers and the handles and pointers to this array type make life much easier.

Pascal

```
TYPE
        SICN          = ARRAY[0 .. 15] of INTEGER;
        SICNList      = ARRAY[0 .. 0] of SICN;
        SICNPtr       = ^SICNList;
        SICNHand      = ^SICNPtr;
```

C

```
typedef    short        SICN[16];
typedef    SICN         *SICNList;
typedef    SICNList     *SICNHand;
```

The Missing Count

The 'SICN' resource does not provide a count to indicate the number of small icons contained within; however, you can easily determine this number by dividing the total size of the resource by the size of a single small icon.

Pascal

```
CONST
       mySICN      = 1984;
VAR
       theSICN     : SICNHand;
       theSize     : LONGINT;
       theCount    : LONGINT;
       theIndex    : LONGINT;

theSICN := SICNHand(GetResource('SICN', mySICN));
IF (theSICN <> NIL) THEN BEGIN
       theSize := GetHandleSize(Handle(theSICN));
       theCount := theSize DIV sizeof(SICN);
END;
```

C

```
#define mySICN         1984

SICNHand   theSICN;
long       theSize;
long       theCount;
long       theIndex;

theSICN = (SICNHand) GetResource('SICN', mySICN);
if (theSICN) {
       theSize = GetHandleSize((Handle)theSICN);
       theCount = theSize / sizeof(SICN);
}
```

The Plot 'SICN's

The example procedure PlotSICN draws one small icon of a 'SICN' resource. It takes the handle from theSICN and the position in the list from theIndex within the rectangle theRect of the current GrafPort.

Following is an example call to PlotSICN which plots all the small icons in a resource into the same rectangle:

Pascal

```
SetRect(theRect, 0, 0, 16, 16);
FOR theIndex := 0 TO theCount-1 DO
       PlotSICN(theRect, theSICN, theIndex);
```

C

```
SetRect(&theRect, 0, 0, 16, 16);
for (theIndex = 0; theIndex < theCount ; ++theIndex)
       PlotSICN(&theRect, theSICN, theIndex);
```

Because PlotSICN uses _CopyBits and _CopyBits can move memory, you should lock the handle to the 'SICN' once the resource is loaded. Notice that the PlotSICN procedure dereferences the 'SICN' handle, adds an offset, and copies the resulting value. If the 'SICN' list moves in memory at this time, the bitmap's baseAddr is useless.

To play it safe, `PlotSICN` saves a copy of the master pointer flags associated with the relocatable block, locks the block with a call to `_HLock`, and restores the flags after calling `_CopyBits`. You should **never** examine, set, or clear these flags directly; you should always use the routines which are provided by the Memory Manager and Resource Manager. Note that it is not necessary to check the value of the flag after getting it.

Pascal

```
PROCEDURE PlotSICN(theRect: Rect; theSICN: SICNHand; theIndex : INTEGER);
VAR
       state       : SignedByte;    { we want a chance to restore original state }
       srcBits     : BitMap;        { built up around 'SICN' data so we can _CopyBits }

BEGIN
       { check the index for a valid value }
       IF (GetHandleSize(Handle(theSICN)) DIV sizeof(SICN)) > theIndex THEN
       BEGIN

           { store the resource's current locked/unlocked condition }
           state := HGetState(Handle(theSICN));

           { lock the resource so it won't move during the _CopyBits call }
           HLock(Handle(theSICN));

           { set up the small icon's bitmap }
           {$PUSH}
           {$R-}                    { turn off range checking }
           srcBits.baseAddr := Ptr(@theSICN^^[theIndex]);
           {$POP}
           srcBits.rowBytes := 2;
           SetRect(srcBits.bounds, 0, 0, 16, 16);

           { draw the small icon in the current grafport }
           CopyBits(srcBits,thePort^.portBits,srcBits.bounds,theRect,srcCopy,NIL);

           { restore the resource's locked/unlocked condition }
           HSetState(Handle(theSICN), state);
       END;
END;
```

C

```
void PlotSICN(Rect *theRect, SICNHand theSICN, long theIndex) {
       auto    char    state;       /* saves original flags of 'SICN' handle */
       auto    BitMap    srcBits;   /* built up around 'SICN' data so we can _CopyBits */

       /* check the index for a valid value */
       if ((GetHandleSize(Handle(theSICN)) / sizeof(SICN)) > theIndex) {

           /* store the resource's current locked/unlocked condition */
           state = HGetState((Handle)theSICN);

           /* lock the resource so it won't move during the _CopyBits call */
           HLock((Handle)theSICN);

           /* set up the small icon's bitmap */
           srcBits.baseAddr = (Ptr) (*theSICN)[theIndex];
           srcBits.rowBytes = 2;
           SetRect(&srcBits.bounds, 0, 0, 16, 16);

           /* draw the small icon in the current grafport */
           CopyBits(&srcBits,&(*qd.thePort).portBits,&srcBits.bounds,theRect,srcCopy,nil);

           /* restore the resource's locked/unlocked condition */
           HSetState((Handle) theSICN, state);
       }
}
```

That Was Easy

Now that you've seen it done, it looks pretty easy. With minor modifications, some of the techniques in this Note could also be used to plot a bitmap of any dimension.

Further Reference:

- *Inside Macintosh*, Volume I, QuickDraw
- *Inside Macintosh*, Volume I, Toolbox Utilities
- *Inside Macintosh*, Volume IV, The Memory Manager
- Technical Note #41, Drawing Into an Off-Screen BitMap
- Technical Note #55, Drawing Icons

#253: 'SICN' Tired of Large Icons in Menus?

Revised by: Dennis Hescox October 1989
Written by: Dennis Hescox August 1989

This Technical Note describes a new facility of the Menu Manager which allows you to add reduced icons and small icons to your menus.
Changes since August 1989: Corrected references to SetItemCmd from SetItmCmd.

Since the release of MultiFinder, you may have noticed the appearance of small icons ('SICN') in the menus of some System Software. At that time, the Menu Manager was modified to allow the capability of showing both 'SICN' resources and 'ICON' resources reduced to 'SICN' size.

How to Add Less

To add one of the smaller icons to a menu item with Rez or ResEdit, do the following:

Reduced Icon

- Place a value of $1D into the cmdChr field of the menuItem.
- Place the resource ID number of the 'ICON' to use, minus 256, into the itemIcon field of the menuItem.

Small Icon

- Place a value of $1E into the cmdChr field of the menuItem.
- Place the resource ID number of the 'SICN' to use, minus 256, into the itemIcon field of the menuItem.

In the ResEdit 'MENU' template, the cmdChr field is called “Key equiv” and the itemIcon field is called “Icon#.”

For setting or changing the menu from within your program, use the following:

```
SetItemCmd(theMenu,item,$1D)          { mark menu item as having a reduced icon }
SetItemIcon(theMenu,item,icon)
```

or

```
SetItemCmd(theMenu,item,$1E)          { mark menu item as having a SICN }
SetItemIcon(theMenu,item,icon)
```

Note that the resource ID that you indicate to the Menu Manager is 256 less than the icon's real resource ID. This means that you can only use icons starting with resource ID of 257 (remember that a zero indicates no icon). Figure 1 illustrates a menu with `'SICN'` resources in the first three items, a normal `'ICON'` in the fourth item, and a reduced version of the normal `'ICON'` in the fifth item.

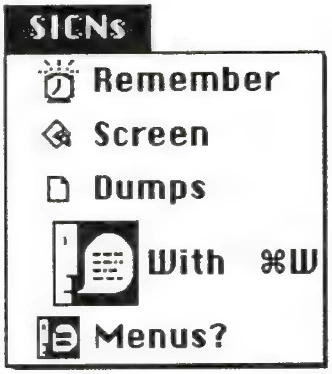


Figure 1–Menu Containing a 'SICN', an 'ICON', and a Reduced 'ICON'

You Win Some; You Lose Some

Note that this new facility does not come for free. A menu item that contains a `'SICN'` or a reduced icon cannot also have a command key equivalent. Because the addition of a smaller icon must be somehow recorded into the existing menu record, the `cmdChr` field of your menu item that used to contain the command key equivalent is now used to indicate both the command key to use or the use of a smaller icon.

Further Reference:

- *Inside Macintosh*, Volume I, The Menu Manager
- Inside Macintosh Volume V, The Menu Manager

Macintosh
Technical Notes

Developer Technical Support

#254: Macintosh Portable PDS Development

Revised by: Dennis Hescox — February 1990
Written by: Dennis Hescox — October 1989

The Technical Note describes the unique aspects of the Macintosh Portable Processor Direct Slot (PDS), including the severe limitations in its use.
Changes since October 1989: Corrected PDS pin and signal descriptions in Tables 2 and 3.

The internal operating environment of the Macintosh Portable is unique within the Macintosh family due to the additional design goals that are not normally applied to other Macintoshes. In particular, two of these goals which limit the use of the PDS are that the unit shall have a long (eight hour) battery operation life and that the unit shall meet all FCC regulations, including the ability to operate on commercial aircraft.

I've Got a Bad Feeling About This

Because of these design goals and the subsequent limitations on the use of the PDS, you must severely limit your card design for the Macintosh Portable.

The first and foremost limitation is that the PDS has **no power budget** for your card. Seeing that there are +12V and +5V connections on the PDS connector, we all realize that you could draw some power directly from the Macintosh Portable. Please don't do it. Instead, you should add your own power supply (i.e., battery) to your board, thus controlling your own destiny (or at least the destiny of your PDS board) and ensuring that the Macintosh Portable has the longest battery life of any portable on the market. You are the best judge as to whether or not your board needs to run continuously when the Macintosh Portable is in sleep mode, therefore requiring a long current life. You might find that the functionality of your board is only optimal when the Macintosh Portable is in full-operating mode (or powered by an external source), and in this case, you could conserve its current demands.

For those of you who are convinced that your product is so important that your users will overlook a 50% reduction in their system operating time, Table 1 shows a worst-case power budget that could apply.

Power Supply	Operating state	Sleep State
+5 V, always on	50 mA maximum	1 mA maximum
+5 V, switched	•	0 mA maximum
+12 V	25 mA maximum	0 mA maximum

- The 50 mA maximum applies to the loads of the switched and unswitched +5 V supplies.

Table 1–Worst-Case Power Budget

The second limitation is that to meet FCC limits on radio frequency emissions, no connector or cable attached to an expansion card can penetrate the case of the Macintosh Portable.

So Why Have a PDS Connector at All?

The decision to include the PDS connector is a recognition that we can't know it all. Although it may seem that next to no power availability and absolutely no custom cables to the outside world would block all possible products, providing the expansion connector allows for that spark of genius for which developers are known and the unanticipated product which usually results. So, if after all these dire warnings you still want to proceed, following are the available details (at least until *Designing Cards and Drivers for the Macintosh* can be updated).

Hang On

The PDS in the Macintosh Portable provides the microprocessor address, control, data, clock power, and Macintosh Portable-specific lines for your expansion card's use. Table 2 lists these signals, while Table 3 lists their descriptions.

Pin Number	Row A	Row B	Row C
1	GND	GND	GND
2	+5V	+5V	+5V
3	+5V	+5V	+5V
4	+5V	+5V	+5V
5	/DELAY.CS	/SYS.PWR	/VPA
6	/VMA	/BR	/BGACK
7	/BG	/DTACK	R/W
8	/LDS	/UDS	/AS
9	GND	+5/0V	A1
10	A2	A3	A4
11	A5	A6	A7
12	A8	A9	A10
13	A11	A12	A13
14	A14	A15	A16
15	A17	A18	reserved
16	reserved	reserved	nc
17	nc	reserved	reserved
18	reserved	reserved	reserved
19	reserved	+12V	D0
20	D1	D2	D3
21	D4	D5	D6
22	D7	D8	D9
23	D10	D11	D12
24	D13	D14	D15
25	+5/3.7V	+5V	GND
26	A19	A20	A21
27	A22	A23	E
28	FC0	FC1	FC2
29	/IPL0	/IPL1	/IPL2
30	/BERR	/EXT.DTACK	/SYS.RST
31	GND	16M	GND
32	GND	GND	GND

Table 2–Macintosh Portable 68000 Direct Slot Expansion Connector Pinouts

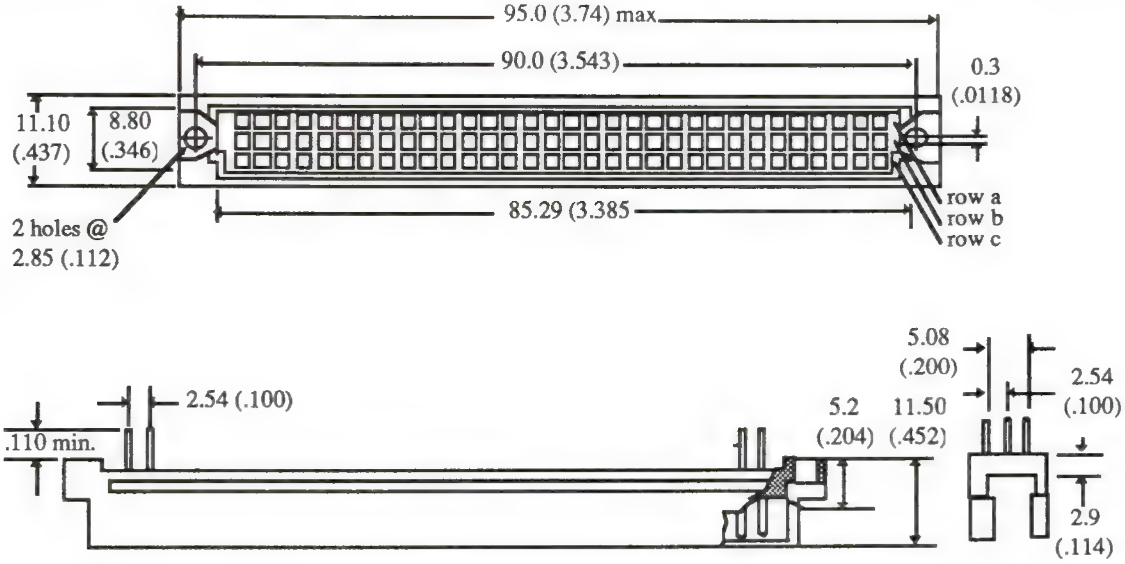
Mnemonic	Description
nc	No connection
GND	Logic ground
D0-D15	Unbuffered data bus, bits 0 through 15
A1-A23	Unbuffered address bus, bits 1 through 23
16M	16 MHz clock
/EXT.DTACK	External data transfer acknowledge. This signal is an input to the processor logic glue. Assertion delays external generation of the /DTACK signal.
E	E (enable) clock
/BERR	Bus error signal generated whenever /AS remains low for more than about 250 µs
/IPL0-/IPL2	Input priority level lines 0 through 2.
/SYS.RST	Initiates a system reset.
/SYS.PWR	A signal from the Power Manager indicating that associated circuits should tri-state their outputs and go into idle state; /SYS.PWR is pulled high (deasserted) during sleep state.
/AS	Address strobe
/UDS	Upper data strobe
/LDS	Lower data strobe
R/W	Defines bus transfer as read or write signal
/DTACK	Data transfer acknowledge
/DELAY.CS	Indicates that a wait state is inserted into the current memory cycle and that you can delay a CS.
/BG	Bus grant
/BGACK	Bus grant acknowledge
/BR	Bus request
/VMA	Valid memory access
/VPA	Valid peripheral address
FC0-FC2	Function code lines 0 through 2
+5/0V	Provides +5V when the system is running normally and 0V when the system is in sleep mode.
+5V/3.7V	Provides +5V when the system is running normally and 3.7V when the system is in sleep mode.

Table 3–Functional Description of the Macintosh Portable PDS Signals

The signals listed in Tables 2 and 3 are presented to your PDS card through a Euro-DIN 96-pin socket connector on the main logic board.

Currently, you can order these Euro-DIN 96-pin connectors (which meet Apple specifications) from: AMP Incorporated, Harrisburg, PA 17105.

Disclaimer: This listing for AMP Incorporated neither implies nor constitutes an endorsement by Apple Computer, Inc. If your company supplies these connectors and you would like to be listed, contact DTS at the address in Technical Note #0.



Three-row pin connector
96 contact positions
2.54 mm (.100) spacing pins
Gold plated, 20 microinches, over nickel plate

Dimensions are in millimeters with inches in parentheses.

Figure 1–96-Pin Plug Connector

Due to the limited space within the Macintosh Portable's case, your card is limited to the size indicated in Figure 2. Apple highly recommends the use of CMOS circuits to reduce the total power necessary for your card's operation.

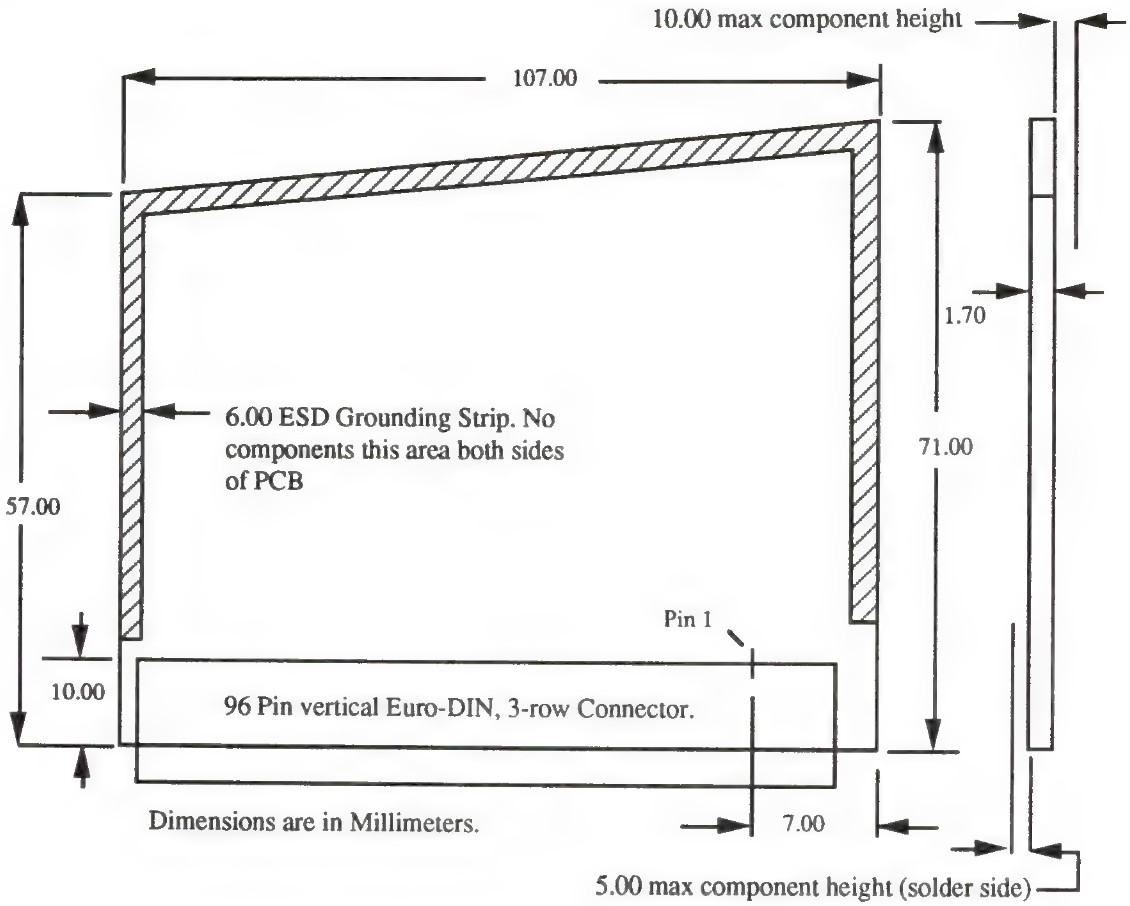

10.00 max component height
107.00
1.70
6.00 ESD Grounding Strip. No components this area both sides of PCB
71.00
57.00
Pin 1
10.00
96 Pin vertical Euro-DIN, 3-row Connector.
Dimensions are in Millimeters.
7.00
5.00 max component height (solder side)

Figure 2–PDS Expansion Card Dimensions

Further Reference:

- *Designing Cards and Drivers for the Macintosh*
- *Guide to the Macintosh Family Hardware*

Macintosh
Technical Notes

Developer Technical Support

#255: Macintosh Portable ROM Expansion

Written by: Dennis Hescox October 1989

This Technical Note explains the practice of and theory behind compatible use of the expansion ROM in the Macintosh Portable.

Due to the unique nature of the Macintosh Portable, developers now have the ability to add ROM to the Macintosh. To provide for compatible shared use of this ROM space with Apple and other developers, this Note describes the feature and suggests methods of shared implementation.

Address Space

The Macintosh Portable contains 256K of processor ROM, which is fundamentally the same as the ROM in the Macintosh SE. This ROM is located at the low end of a 1 MB ROM space. With an expansion card, one can either completely replace the 1 MB ROM or simply add an additional 4 MB of ROM. The original 1 MB of address space is **reserved** for use by Apple, but the additional 4 MB address space is available for third-party developers.

Apple reserved ROM space is located from $90 0000 through $9F FFFF. You can replace this ROM space with an expansion board, thus overriding these ROMs; however, if you override these ROMs your machine will no longer work with most applications. This ability to override the original ROMs is intended for Apple in the event that a ROM upgrade is ever necessary for the Macintosh Portable. Developers should use the 4 MB ROM address space from $A0 0000 through $DF FFFF, which is illustrated in Figure 1, for expansion.

Since Apple could provide a ROM upgrade (on a ROM expansion board), we recommend that developers use a standard 32-pin DIP socketed ROM part for any expansion board. Following this recommendation ensures that the user will never have to choose between an Apple ROM upgrade and a third-party expansion board, since Apple could provide sockets for third-party ROMs if we were to produce such an upgrade.

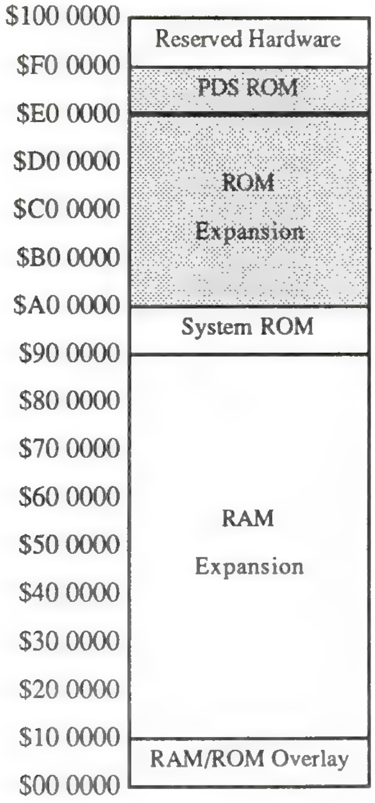


Figure 1–Macintosh Portable Memory Map

Expansion ROM Board

If Apple were to produce an expansion ROM board for an upgrade, it would have the following characteristics. Side one would contain four 32-pin ROM sockets compatible with 128K x 8 bit or 512K x 8 bit ROMs, a dip switch for choosing between 128K or 512K socket address sizes, and appropriate decoupling capacitors. Side two would contain Apple's expansion ROMs and any additional circuitry. This design implies that developers would be able to use at most either 512K or 2 MB of the total 4 MB expansion space.

When designing your own expansion board, remember that it must contain circuitry for decoding, controlling, and buffering, and it should use CMOS, since the Macintosh Portable restricts ROM expansion boards to a maximum of 25ma. The number of wait states inserted depends upon the DTACK generated by your board, which connects to the Macintosh Portable through a single 50-pin connector (slot). The machine provides all of the appropriate signals (address bus, data bus, and control) to the expansion slot, where they are decoded into chip selects and routed to address and data buffers. These signal names and descriptions are illustrated in Figure 2 and described in Table 1. It is also important to buffer the address and data buffers to reduce capacitive loading.

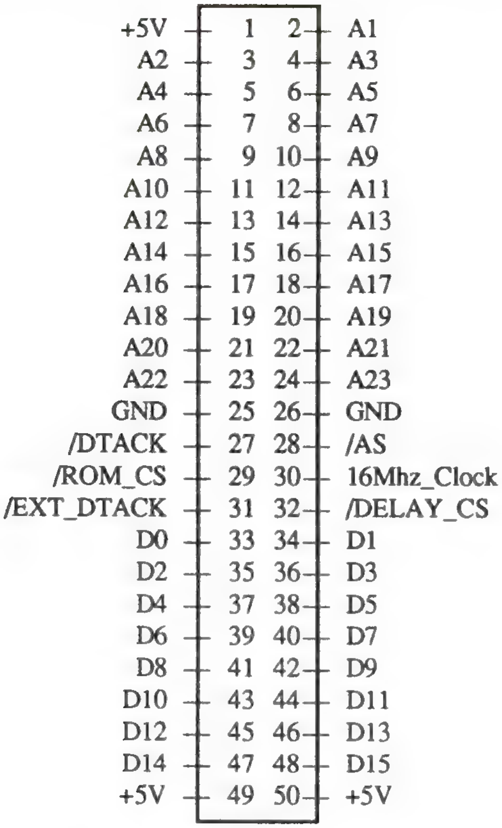
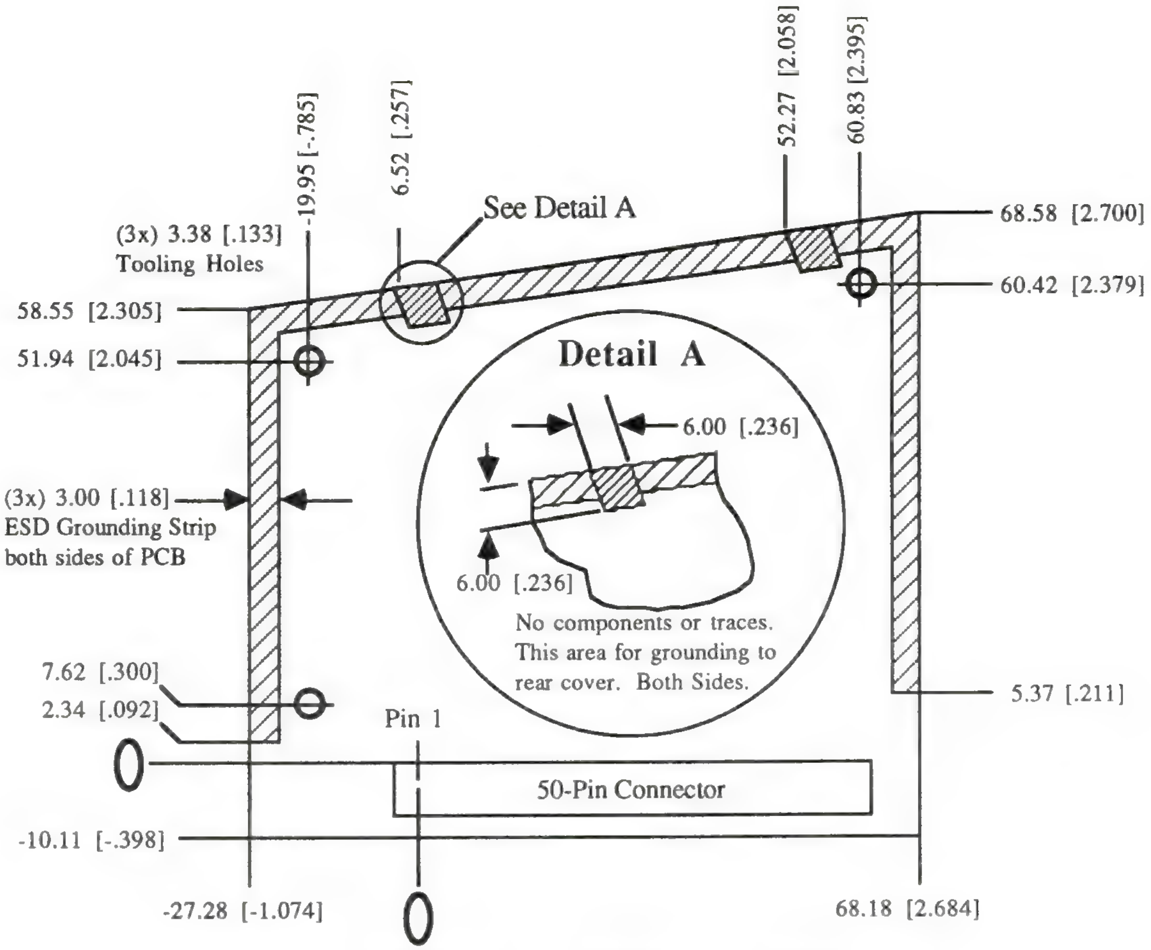


Figure 2–Internal ROM Expansion Connector Signals

Pin Number	Signal Name	Signal Description
1	+5V	Vcc
2-24	A1-23	Unbuffered 68HC000 address signals A1-23
25-26	GND	Logic Ground
27	/DTACK	/DTACK input to 68HC000
28	/AS	68HC000 address strobe signal
29	/ROM_CS	Permanent ROM chip select signal. Selects in range $90 0000 through $9F FFFF.
30	16 Mhz_clock	16 Mhz system clock.
31	/EXT_DTACK	External /DTACK signal that disables main system /DTACK
32	/DELAY_CS	This signal is generated by the addressing PAL and is used to put the ROM board into the idle mode by inserting multiple wait states.
33-48	D0-15	68HC000 unbuffered data signals D0-15
49-50	+5V	Vcc

Table 1–Internal ROM Expansion Connector Signal Descriptions



Dimensions are in Millimeters [Inches]

Figure 3–Internal ROM Expansion Board Guidelines

Software Standards

For the purposes of expansion ROM, Apple has introduced Electronic Disks (EDisks), which appear to the user as very fast, silent disk drives. The EDisk driver supports EDisks, which use RAM or ROM as their storage media.

ROM EDisks, which can be produced by third parties, are connected to the system using the internal ROM expansion slot. The 4 MB address space allocated for this type of expansion supports any number of ROM EDisks, as long as they start on a 64K boundary (their size may exceed 64K). ROM EDisks behave like RAM EDisks, except that they are read-only and cannot be resized.

The EDisk Driver

The EDisk driver provides a system interface to EDisks similar to that provided by the Sony and SCSI disk drivers. It supports 512 byte block I/O operations and does not support file system tags. The EDisk driver is a ROM `'DRVR'` resource with an ID of 48, `RefNum` of -49, and driver name of ".EDisk". Since it is a disk driver, it also creates a Drive Queue Element for each EDisk. Information on how these driver calls apply to the Sony driver appear in the Disk Driver chapters of *Inside Macintosh*, Volumes II, IV, & V.

EDisk Implementation Details

The remainder of this section describes some of the implementation details, data formats, and algorithms used by the EDisk driver that may be useful for developers who want to produce ROM EDisks.

Data Checksumming

To provide better data integrity, the EDisk driver supports checksumming of each data block, which is computed when a write is performed to a block and checked on every read operation. It computes a 32-bit checksum for each 512-byte block. This calculation is performed by adding each longword in the block to a running longword checksum, which is initially zero, and is rotated left by one bit before each longword is added. The following assembly code demonstrates this algorithm:

```
        Lea         TheBlock,a0      ; A0 is pointer to the block to checksum
        Moveq.L     #0,D0            ; D0 is the checksum, initially zero
        Moveq.L     #(512/4)-1,D1    ; loop counter for 1 block (4 bytes per iteration)
@Loop   Rol.L       #1,D0            ; rotate the checksum
        Add.L       (A0)+,D0         ; add the data to the running checksum
        Dbra        D1,@Loop         ; loop through each longword in the block
```

Internal ROM EDisk Details

When the EDisk driver is opened, it searches the address range from the base of the system ROM to $00E0 0000 for internal ROM EDisks. An internal ROM EDisk must begin with an EDisk header block, which must start on a 64K boundary (but may be any size). If a valid header block is found, it is compared to all other known headers, and if it is identical to another, it is ignored to eliminate duplicates caused by address wrapping. If the header block is unique, the EDisk driver supports it and creates a drive queue entry for it. The driver can support any number of internal ROM EDisks, and it is limited only by the address space allocated for ROM.

EDisk Header Format

There is a 512-byte header block associated with ROM EDisks. This header describes the layout of the EDisk and uniquely identifies it. The general format of the header block is described below. The EDisk header marks the beginning of an EDisk, and it should occur at the beginning of the ROM space that is used for EDisk storage (i.e., starting at the first byte of a 64K ROM block).

```
EDiskHeader    Record 0,increment    ; layout of the EDisk signature block
HdrScratch            DS.B    128    ; scratch space for r/w testing and vendor info
HdrBlockSize          DS.W    1      ; size of header block (512 bytes for version 1)
HdrVersion            DS.W    1      ; header version number (this is version 1)
HdrSignature          DS.B    12     ; 45 44 69 73 6B 20 47 61 72 79 20 44
HdrDeviceSize         DS.L    1      ; size of device, in bytes
HdrFormatTime         DS.L    1      ; time when last formatted (pseudo unique ID)
```

```
HdrFormatTicks      DS.L    1       ; ticks when last formatted (pseudo unique ID)
HdrCheckSumOff      DS.L    1       ; offset to the Checksum table, if present
HdrDataStartOff     DS.L    1       ; offset to the first byte of data storage
HdrDataEndOff       DS.L    1       ; offset to the last byte+1 of data storage
HdrMediaIconOff     DS.L    1       ; offset to the media Icon and Mask, if present
HdrDriveIconOff     DS.L    1       ; offset to the drive Icon and Mask, if present
HdrWhereStrOff      DS.L    1       ; offset to the  Get Info Where: string, if present
HdrDriveInfo        DS.L    1       ; longword for Return Drive Info call, if present
                    DS.B    512-*   ; rest of block is reserved
EDiskHeaderSize     EQU     *       ; size of EDisk header block
                    ENDR
```

`HdrScratch` is a 128-byte field that is used for read and write testing on RAM EDisks to determine if the memory is ROM or RAM. On ROM EDisks, it should be filled in by the vendor with a unique string to identify this version of the ROM EDisk (e.g., “Copyright 1989, Apple Computer, Inc. System Tools 6.0.4 9/5/89”).

`HdrBlockSize` is a 2-byte field that indicates the size of the EDisk header block. The size is currently 512 bytes.

`HdrVersion` is a 2-byte field that indicates the version of the EDisk header block. The version number is currently $0001.

`HdrSignature` is a 12-byte field that identifies a valid EDisk header block. The signature must be set to `45 44 69 73 6B 20 47 61 72 79 20 44` in hexadecimal.

`HdrDeviceSize` is a 4-byte field that indicates the size of the device in bytes, which may be greater than the actual usable storage space. One might also think of the device size as the offset (from the beginning of the header block) of the last byte of the storage device.

`HdrFormatTime` is a 4-byte field that indicates the time of day when the EDisk was last formatted. The EDisk driver updates this for RAM EDisks when the format control call is made. This information may be useful for uniquely identifying a RAM EDisk.

`HdrFormatTicks` is a 4-byte field that indicates the value of the system global `Ticks` when the EDisk was last formatted, which should be a unique number. The EDisk driver updates this for RAM EDisks when the format control call is made. This information may be useful for uniquely identifying a RAM EDisk.

`HdrCheckSumOff` is a 4-byte field that is the offset (from the beginning of the header block) of the checksum table, or zero if checksumming should not be performed on this EDisk.

`HdrDataStartOff` is a 4-byte field that is the offset (from the beginning of the header block) of the first block of EDisk data.

`HdrDataEndOff` is a 4-byte field that is the offset (from the beginning of the header block) of the byte after the end of the last block of EDisk data.

`HdrMediaIconOff` is a 4-byte field that is the offset (from the beginning of the header block) of the 128-byte icon and 128-byte icon mask, which represents the disk media. An offset of zero indicates that the EDisk driver should use the default media icon for this EDisk.

`HdrDriveIconOff` is a 4-byte field that is the offset (from the beginning of the header block) of the 128-byte icon and 128-byte icon mask, which represents the disk drive physical location. An offset of zero indicates that the EDisk driver should use the default drive icon for this EDisk.

`HdrWhereStrOff` is a 4-byte field that is the offset (from the beginning of the header block) of the Pascal string that describes the disk location for the Finder Get Info command. An offset of zero indicates that the EDisk driver should use the default string for this EDisk.

`HdrDriveInfo` is a 4-byte field that should be returned by the drive information control call. A value of zero indicates that the EDisk driver should use the default drive info for this EDisk.

You should not override the default media or drive icons without first giving serious consideration as to how a different icon will affect the user interface. What often appears to be a clever idea for a cute icon usually turns out to be a source of frustration for the user when deciding what the item is and where it is physically located.

Some Final Thoughts

Do Not Use More Space Than You Need

As wonderful and indispensable as your ROM product may be, users may wish to also use ROMs from another developer. Although ROM address space is quite large (in today's terms), board space and number of ROM chip sockets is limited. If you use only the space you really need and leave room (address space and empty chip sockets) in your ROM product to add other ROMs, users will never have to make a choice between your product and another, unanticipated stroke of genius.

Keep It Relocatable

Just because your code is in ROM does not mean that it will always reside at a specific address. When moving your ROM to another board (an Apple upgrade or another third-party board), users should neither have to worry about address range conflicts nor socket location. In addition, Apple may implement ROM expansion in a future product with expanded or different address space; keeping your ROM code relocatable could mean the difference between additional sales or incompatibility and upgrades.

Further Reference:

- *Inside Macintosh*, Volume II, IV, & V, The Disk Driver

Macintosh
Technical Notes

Developer Technical Support

#256: Stand-Alone Code, *ad nauseam*

Written by: Craig Prouse
Inspired by: Keith Rollin & Keithen Hayenga

August 1990
October 1989

This Technical Note discusses many of the issues related to stand-alone code modules. This Note is by no means a completely original work, as the author borrows freely from the work of Keith Rollin, Mark Baumwell, and Jim Friedlander.
Changes since October 1989: Completely rewritten to broaden the discussion of stand-alone code modules and include a greater scope of examples. Incorporates Technical Notes #110, MPW: Writing Stand-Alone Code and #145, Debugger FKEY.

How to Recognize a Stand-Alone Code When You See One

What Stand-Alone Code Looks Like to the Naked Eye

Stand-alone code is program code which does not enjoy the full status of an application. A stand-alone code module exists as a single Macintosh resource and consists almost entirely of microprocessor-executable object code, and perhaps also some header data and other constants used by the executable portion of the module. Code-type resources are most easily identifiable in the ResEdit 2.0 resource picker. Most of these closely-related resources are indicated by an icon containing a stylized segment of assembly-language source code.

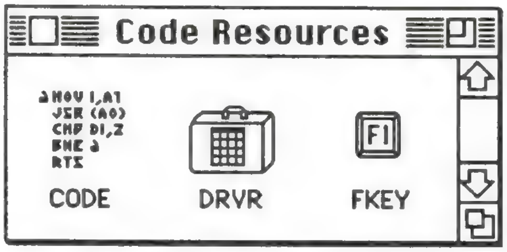


Figure 1–ResEdit 2.0 Icons Signifying Code-Type Resources

Although `'CODE'` resources are not stand-alone code modules (they are segments of a larger application), they are similar because they contain executable code and so they have the same code-type icon. Driver resources are a special case of stand-alone code resources, and they have a different icon in the ResEdit picker, reminiscent of the Finder icon for a desk accessory suitcase, because the code of a desk accessory is stored as a `'DRVR'` resource. The icon for an `'FKEY'` is also a bit different, resembling a function key, naturally.

Table 1 is a fairly extensive list of the currently-defined code-type resources. Many are of interest primarily at a system software level; those stand-alone code resources most commonly created by application-level programmers are noted in boldface. Of course, developers are always free to define new resource types for custom stand-alone modules. `'CUST'` is commonly used, as in some of the examples at the end of the discussion.

ADBS	adev	CACH	**CDEF**	**cdev**	CODE	dcmd
DRVR	FKEY	FMTR	**INIT**	itl2	itl4	**LDEF**
MBDF	**MDEF**	mntr	PACK	PDEF	PTCH	ptch
rdev	ROvr	RSSC	snth	**WDEF**	**XCMD**	**XFCN**

Table 1–Assorted Code Resource Types

The most common use of stand-alone code is to supplement the standard features provided by the Macintosh Toolbox and operating system. Most of the resource types listed in Table 1 define custom windows, controls, menus, lists, and responses to user input. In this respect, they are slaves to particular Toolbox managers or packages and very often contained within the resource fork of an owner application. Other examples of stand-alone code are more useful as application extensions like HyperCard `'XCMD'` and `'XFCN'` extensions.

`'DRVR'`, `'INIT'`, and `'cdev'` resources are more autonomous examples of stand-alone code. These allow programmers to write code which may be executed automatically when the system starts up and code which adds special features to the operating system or provides control of special-purpose peripherals and system functions. The temptation here is to perform functions generally reserved for full-blown applications, such as use of QuickDraw. For a number of reasons, this is a non-trivial endeavor, and is the subject of much of this discussion.

How Applications Are Special

Macintosh applications can be almost any size, limited mainly by disk space and RAM size. The actual program code is generally divided up into a number of segments, each less than 32K in size so the amount of memory required to execute a program may be less than the size of the program itself. The Segment Loader, documented in *Inside Macintosh*, Volume II, controls the loading and unloading of segments. It ensures that the necessary segments of an application are read into the application heap when needed and allows temporarily unneeded sections to be purged, making room for others.

All of this activity occurs in and depends upon a somewhat mysterious construction called an `A5` world. It is so called because the `A5` register of the microprocessor points indirectly to several key data structures used by the Segment Loader and the application itself. Most Macintosh programmers are at least vaguely aware of the significance of `A5` in the Macintosh environment. Many even know that it is a handy pointer to the application and QuickDraw global variables, or at least points in the right general direction. Less widely known is how an `A5` world is constructed, and more to the point, how to build one from scratch if necessary.

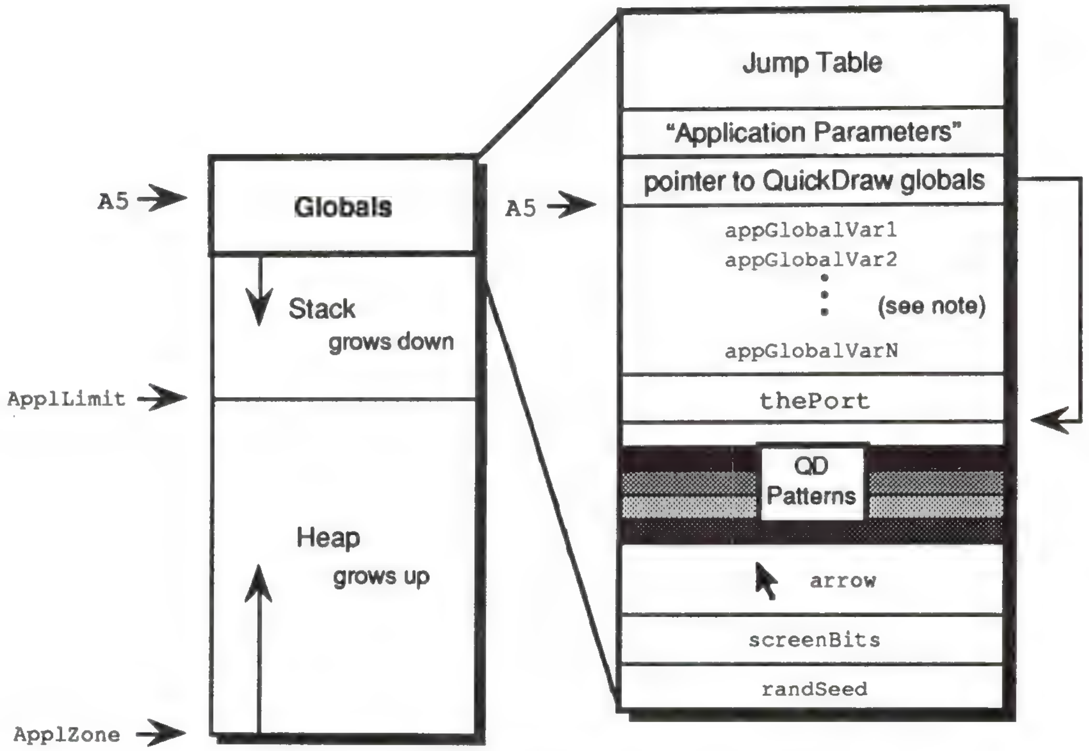
This may become necessary because higher-level language compilers like MPW Pascal and C automatically generate `A5`-relative addressing modes to refer to global variables, including QuickDraw globals. The linker then resolves the actual offsets. For example, the ubiquitous

```
    InitGraf(@thePort);        {initialize QuickDraw}
```

compiles into something equivalent to the following:

```
    PEA   thePort(A5),-(SP)    ; push a pointer to QuickDraw's thePort variable
    _InitGraf                  ; invoke _InitGraf trap to initialize QuickDraw
```

Before this is executable, the linker must determine exactly what offset represents `thePort`. With this value, it patches the object code and creates the code found in the final application. The reader may infer that an application depends on someone else to set up `A5` with a meaningful value before program execution begins. This is true, and understanding how this process normally occurs for an application is of paramount importance when writing stand-alone code which needs application-like functionality. Briefly, the Segment Loader allocates space for an `A5` world for each application as part of its launch process. Library code is automatically linked to the front of every application, and this sets up `A5` to point to the global variable space. The application code begins executing only after all of this preliminary setup is complete.



Note: Application globals may appear above or below the QuickDraw globals. This is linker-dependent. What's important is that separately-linked external modules can use A5 to locate an application's QuickDraw globals.

Figure 2–A Hitchhiker's Guide to the A5 World

How Stand-Alone Code Is Different

Stand-alone code, unlike an application, is never launched. It is simply loaded then executed and possesses no `A5` world of its own. Stand-alone code therefore cannot easily define global variables. No space is allocated for globals and `A5` either belongs to a real application or is completely meaningless. References to global variables defined by the module usually succeed without even a warning from the linker, but also generally overwrite globals defined by the current application. References to global variables defined in the MPW libraries, like QuickDraw globals, generate fatal link errors.

```
Link -t INIT -c '????' -rt INIT=128 -ra =resLocked -m PLAYZOO ∂
    SampleINIT.p.o ∂
    -o SampleINIT
### Link: Error: Undefined entry, name: (Error 28) "thePort"
  Referenced from: PLAYZOO in file: SampleINIT.p.o
### Link: Errors prevented normal completion.
### MPW Shell - Execution of SampleINIT.makeout terminated.
### MPW Shell - Execution of BuildProgram terminated.
```

That's not very helpful and not very much fun. So what if a stand-alone code resource needs to use QuickDraw or its associated globals like `screenBits`? What if a stand-alone module needs to call some "innocuous" routine in the Macintosh Toolbox which implicitly assumes the existence of a valid `A5` world? `_Unique1ID`, which calls the QuickDraw `_Random` trap, falls into this category, for instance. An `'XCMD'` might be able to "borrow" HyperCard's globals, but an `'INIT'` has no such alternative; it may need to have its own `A5` world.

There are a couple more considerations. Stand-alone code resources are not applications and are not managed by the Segment Loader, so they cannot be segmented into multiple resources like applications. Stand-alone code resources are self-contained modules and are usually less than 32K in size. As popular belief would have it, code resources **cannot** be more than 32K in size. This is not necessarily true, and although some linkers, especially older ones, enforce the limit all the same, the absolute limitation is that the original Motorola MC68000 microprocessor is not capable of expressing relative offsets which span more than 32K.

A code segment for a 68000-based Macintosh may be any reasonable length, so long as no relative offsets exceed 32K. There are ways to get around this limit even on 68000-based machines, while the MC68020 and later members of the 680x0 family have the ability to specify 32-bit offsets, dissolving the 32K barrier completely as long as the compiler is agreeable. To remain compatible with 68000-based machines, however, and to maintain manageable-sized code segments the 32K "limit" is a good rule of thumb. If a stand-alone code module gets much larger than this, it is often because it's trying to do too much. Remember that stand-alone code should only perform simple and specific tasks.

Writing Your First Stand-Alone Module

Each type of stand-alone code has its own idiosyncrasies. It is difficult to say which type is the easiest to construct. It is best to address each major type individually, but a simple `'INIT'` may be the best place to start. Most programmers are pretty familiar with the concept of what an `'INIT'` is and how it is used, and its autonomy is a big plus—it is not necessary to write and debug a separate piece of code or a HyperCard stack in which to test the stand-alone module. (This would be necessary for a `'CDEF'` or an `'XCMD'`, for example.) Stand-alone code is often written in assembly language, but high-level languages usually serve just as well. This first example is written in MPW Pascal, in keeping with the precedent set by *Inside Macintosh.*

SampleINIT is a very simple `'INIT'` which plays each of the sounds (resources of type `'snd '`) in the System file while the Macintosh boots. It's kind of fun, not too obnoxious, and also not particularly robust. All `'snd '` resources should be unlocked, purgeable, Format 1 sounds like the four default system sounds. Also be sure to name this file SampleINIT.p to work with the SampleINIT.make file which follows. The main subroutine is `PlayZoo`, in honor of the monkey and dogcow sounds in the author's System file.

```
UNIT SampleINIT;    {Pascal stand-alone code is written as a UNIT}

INTERFACE

  USES
    Types, Resources, Sound;

  {VAR
    cannot define any globals...well, not yet anyway}

  PROCEDURE PlayZoo;

IMPLEMENTATION

  PROCEDURE PlayZoo;
  VAR
    numSnds, i : INTEGER;
    theSnd : Handle;
    playStatus : OSErr;
  BEGIN
    numSnds := CountResources('snd ');
    FOR i := 1 TO numSnds DO BEGIN
      theSnd := GetIndResource('snd ',i);
      IF theSnd <> NIL THEN
        playStatus := SndPlay(NIL,theSnd,FALSE);
      END;
  END;

END.
```

Following is the file SampleINIT.make to control the build process:

```
#   File:       SampleINIT.make
#   Target:     SampleINIT
#   Sources:    SampleINIT.p

SampleINIT ƒƒ SampleINIT.make SampleINIT.p.o
    Link -t INIT -c '????' -rt INIT=128 -ra =resLocked -m PLAYZOO ∂
        SampleINIT.p.o ∂
        -o SampleINIT

SampleINIT.p.o ƒ SampleINIT.make SampleINIT.p
    Pascal SampleINIT.p
```

That's all there is to it, but even in such a simple example as this, there are a number of extremely important points to highlight. By understanding every nuance of this example, one moves a long way toward understanding everything there is to know about stand-alone code.

Consider first the form of the `'INIT'` code itself. It is defined as a `UNIT` rather than a `PROGRAM`. This is important, because Pascal programs are applications and require the Segment Loader, preinitialization of `A5`, and all the things which make an application special. A Pascal unit, like a stand-alone code resource, is simply a collection of subroutines. A similar assembly-language `'INIT'` would define and export a `PROC`. In C, this particular `'INIT'` would be a single function in a source file with no `main()` function.

Pascal programmers may recognize that a unit allows the definition of global variables (as between the `USES` and `PROCEDURE` clauses in the `INTERFACE` section previously documented). Typically, when a unit's object code is linked with a host application, the linker allocates storage for these globals along with the application globals and resolves all necessary `A5` references. Stand-alone code modules are never linked with an application, however, and the linker has no way to resolve these references. This makes the linker very unhappy. The easiest way to make the linker happy is to follow the example and define no globals. If globals are really necessary, and they may well be, read on.

Next examine how SampleINIT is linked. To be recognized as a startup document, a "system extension" (as an `'INIT'` is called in System 7.0 parlance) must have the file type "INIT". The options `-rt` and `-ra` respectively specify that the code resource is of type `'INIT'` (ID=128), and that the resource itself is locked. This is a very important idiosyncrasy of the `'INIT'` because it is not automatically locked when loaded by the system and might otherwise attempt to move during execution. Hint: this would be very bad.

Finally, `PlayZoo` is specified as the main entry point by the `-m` option. When written in Pascal, the entry point of a module is the first compiled instruction. C is a little pickier and demands the main entry point option to override the default entry point (which performs run-time initialization for applications). It is important to remember that the linker does not move the entry point specified by `-m` to the front of the object file—that is the programmer's responsibility. Specification of this option primarily helps the linker remove dead, unused code from the final object module. In short, don't leave home without it. Note that the linker is case sensitive with respect to identifiers, while the Pascal compiler converts them to all uppercase. It is necessary therefore (in this example) to specify the name of the entry point to the linker in all uppercase characters. If `PlayZoo` were written in C, which is also case-sensitive, the identifier would be passed to the linker exactly as it appeared in the source code.

For additional examples of stand-alone code, refer to the end of this Note. There are currently a few examples of types of stand-alone code, some of which illustrate the advanced topics discussed in the following sections.

The next area to investigate is how to get around the restrictions on globals in stand-alone code. The first and simplest solution easily conforms to all compatibility guidelines, and that is to avoid using globals altogether. There often comes a time, however, when the use of a global seems unavoidable. The global variable requirements of stand-alone code segments vary, naturally, and there are a number of possible scenarios. Some involve creating an `A5` world and others do not. It's best to start with the simplest cases, which do not.

Oh, I Have Slipped the Surly Bonds of the Linker . . .

. . . And Have Hung Like a Leach on My Host Application

Often a stand-alone code segment needs the QuickDraw globals of the current application, for whom it is performing a service. A control definition function (`'CDEF'`) is an example. Its drawing operations assume a properly-initialized QuickDraw world, which is graciously provided by the application. Most QuickDraw calls are supported and no special effort is required. One limitation, however, is that explicit references to QuickDraw globals like `thePort` and `screenBits` are not allowed. The linker cannot resolve the offsets to these variables because it does not process a `'CDEF'` (or any other stand-alone module) along with a particular application. Fortunately the solution is simple, if not entirely straightforward.

Since the structure of the QuickDraw global data is known, as is its location relative to `A5`, stand-alone code executing as a servant to an application can reference any desired QuickDraw global indirectly. The following code is an example of how a stand-alone unit can make local copies of all the application QuickDraw globals. It uses `A5` to locate the variables indirectly, rather than making explicit symbolic references which the linker is not capable of resolving. Figure 2, presented earlier, may be helpful in understanding how this code works.

```
UNIT GetQDGlobs;

INTERFACE

  USES
    Types, QuickDraw, OSUtils;

  TYPE
    QDVarRecPtr   =   ^QDVarRec;
    QDVarRec   =   RECORD
              randSeed    : Longint;
              screenBits : BitMap;
              arrow       : Cursor;
              dkGray      : Pattern;
              ltGray      : Pattern;
              gray        : Pattern;
              black       : Pattern;
              white       : Pattern;
              thePort     : GrafPtr;
            END;

  PROCEDURE GetMyQDVars (VAR qdVars: QDVarRec);

IMPLEMENTATION

  PROCEDURE GetMyQDVars (VAR qdVars: QDVarRec);
  TYPE
    LongPtr = ^Longint;
  BEGIN
    { Algorithm:
      1.  Get current value of A5 with SetCurrentA5.
      2.  Dereference to get address of thePort.
      3.  Perform arithmetic to determine address of randSeed.
      4.  By assignment, copy the QD globals into a local data structure. }
    qdVars := QDVarRecPtr(LongPtr(SetCurrentA5)^ - (SizeOf(QDVarRec)-SizeOf(thePort)))^;
  END;

END.
```

Extensible Applications

Some applications are intended to be extensible and provide special support for stand-alone code segments. ResEdit for instance, uses `'RSSC'` code resources to provide support for custom resource pickers and editors. If a graphical editor is needed to edit a custom resource type, such as an 8 x 64-pixel icon, separately compiled and linked extension code can be pasted directly into the application's resource fork. ResEdit defines interfaces through which it communicates with these resources. In many cases, this degree of support and message passing can preempt the need to declare global variables at all. The ResEdit interfaces are part of the official ResEdit package available from APDA. The MacsBug `'dcmd'` is another instance of extension code with support for globals built in. A `'dcmd'` specifies in its header how much space it needs for global variables and MacsBug makes room for them.

HyperCard provides high-level support for its `'XCMD'` and `'XFCN'` extension resources. Callback routines like `SetGlobal` and `GetGlobal` provide extension code with a convenient mechanism for defining variables which are global in scope, yet without requiring the deadly `A5`-relative references normally associated with global variables. The HyperCard interfaces are included with MPW and are called HyperXCmd.p in the Pascal world, or HyperXCmd.h for C programmers.

In cases where an application provides special support for extensions, the extension writer should take advantage of this support as much as possible. Things can get complicated quickly when no support for globals is provided or when built-in support is not used, and there's really no reason to be a masochist. The A5-world techniques described later in this Note usually are not necessary and should be considered extraordinary. Also, when writing an application, it is probably worth considering whether extensibility is useful or desirable. With the move toward object-oriented programming and reusable code, demand for extension module support is growing. Support for extension modules can rarely be tacked on as an afterthought, and it is worth looking at how ResEdit, HyperCard, and Apple File Exchange support modular code when considering similar features. Foresight and planning are indispensable.

Calling Stand-Alone Code from Pascal

Before moving on it may be helpful to look at how one extensible application calls stand-alone code, using HyperCard as an example. The first thing to do is establish some standard means of communication. HyperCard uses a clearly-defined parameter block, as defined in HyperXCmd.p.

```
XCmdPtr = ^XCmdBlock;
XCmdBlock = RECORD
  paramCount:  INTEGER;
  params:      ARRAY [1..16] OF Handle;
  returnValue: Handle;
  passFlag:    BOOLEAN;
  entryPoint:  ProcPtr;    {to call back to HyperCard}
  request:     INTEGER;
  result:      INTEGER;
  inArgs:      ARRAY [1..8] OF LONGINT;
  outArgs:     ARRAY [1..4] OF LONGINT;
  END;
```

An 'XCMD' procedure, like an 'INIT', is written, compiled, and linked as a separate unit. Its prototype may be imagined something like this:

```
PROCEDURE MyXCMD (pb: XCMDPtr);
  EXTERNAL;
```

Since MyXCMD is not linked with HyperCard, however, the example declaration does not appear in the HyperCard source code. The prototype only defines how the external module expects to receive its parameters. The host application, HyperCard, is responsible for loading the module and implementing the proper calling conventions.

When calling an 'XCMD', HyperCard first loads the resource into memory and locks it down. It then fills in the parameter block and invokes the 'XCMD'. Notice that the extension module is loaded by its resource name. This is common for extensible applications, since it avoids resource numbering conflicts. Since HyperCard is written in Pascal, the sequence might look something like this.

```
theHandle := Get1NamedResource('XCMD', 'MyXCMD');
HLock(theHandle);
WITH paramBlock DO
  BEGIN
    { fill it in }
  END;
CallXCMD(@paramBlock, theHandle);
HUnlock(theHandle);
```

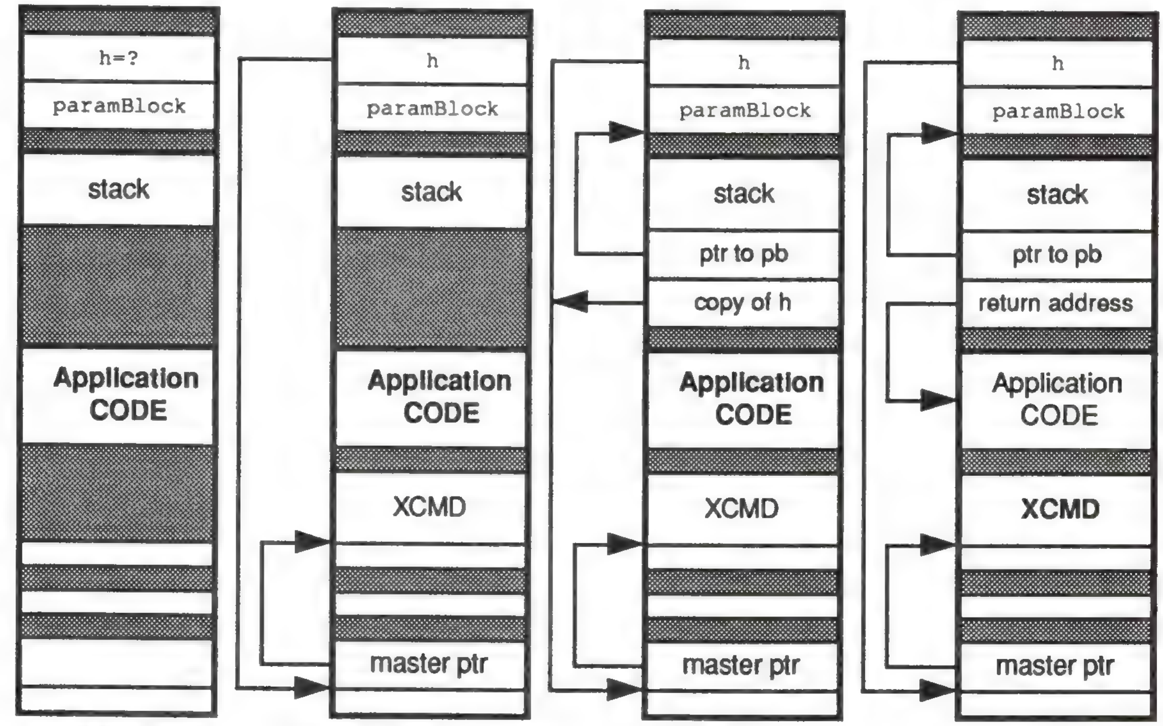
This also looks a little unwieldy. To fully understand a high-level calling sequence for stand-alone code, a working knowledge of parameter passing conventions and the ability to read code at the assembly-language level is very helpful. Some amount of glue code is almost always required, as illustrated by CallXCMD. After Pascal places a pointer to the parameter block and a handle to the

'XCMD' on the stack, it executes some assembly-language glue represented by three inline opcodes. The glue code finds the 'XCMD' in memory and jumps to it using the handle on the stack. To accomplish this, it pulls the handle off of the stack, dereferences it to obtain a pointer to the 'XCMD' and performs a JSR to the indicated address. The pointer to the parameter block is left on the stack for the 'XCMD'.

```
PROCEDURE CallXCMD (pb: XCMDPtr; xcmd: Handle);
  INLINE $205F,     { MOVEA.L  (A7)+,A0         pop handle off stack        }
         $2050,     { MOVEA.L  (A0),A0   dereference to get address of XCMD }
         $4E90;     { JSR      (A0)        call XCMD, leaving pb on stack   }
```

Figure 3 illustrates the state of the A5 world at four critical phases of the 'XCMD' calling sequence. Boldface indicates approximately where the program counter is pointing, or what code is executing at that moment. The easiest way to read the diagram is to look for features which change from one state to the next. Note in the last state the 'XCMD' knows how to find its parameter block because the stack pointer (A7) initially points to the return address and a pointer to the parameter block is located four bytes above that. If the 'XCMD' is written in a high-level language according to the procedure prototype MyXCMD, as shown above, this procedure is handled automatically by the compiler.

The process is essentially the same when calling stand-alone code from assembly language, but it is not so unnatural. The assembly-language programmer never has to leave his element and generally has a better low-level view of where certain data structures reside and how to access them efficiently. Since the entry point of the stand-alone module can be determined directly, there is no exact parallel to the CallXCMD procedure, and it is not necessary to push a copy of the resource handle on the stack as an intermediate step.



Initial State with paramBlock and h as global variables

Get1NamedResource loads 'XCMD' resource into application heap

CallXCMD pushes some stuff on the stack for the glue code

Glue code removes handle from stack and does JSR to address in master pointer

Figure 3–Calling an 'XCMD' from Pascal

Interestingly enough, the `CallXCMD` procedure can be easily modified to call almost any stand-alone module whose entry point is at the beginning of the code resource. To determine the proper calling interface for a particular code module, simply duplicate the function prototype of the module and add a handle at the end of the argument list. The inline glue does not have to change at all. This works equally well for Pascal procedures or functions, and for any number of arguments including `VAR` parameters.

Doing the A5 Road Trip

There comes a time and place where construction of an `A5` world is a "necessary" evil. Usually it's not necessary at all, but sometimes the world really needs just one more Orwellian security `'INIT'` to present a dialog at startup. DTS discourages such things, but they happen. Although there is nothing fundamentally or philosophically wrong with constructing a custom `A5` world, doing so can create significant technical hassles, and unfortunately, globals make possible a number of user interface atrocities. Generally a different solution, if available, results in simpler and more maintainable code, and reduces the likelihood that your code will go the way of the dinosaur and the passenger pigeon. Furthermore, to make the process of constructing an `A5` world as straightforward as possible, yet consistent with normal applications, extensive use is made of two undocumented routines in the MPW run-time libraries. The dangers here are obvious. There are accepted uses, nonetheless. External modules may need to create some global storage or hold data which persists across multiple calls to a routine in the module. All uses shall henceforth be considered fair game, for as it is written in Clarus' memoirs:

Yea, and if It will be done, even in spite,
Then lend Thine hand to the masses,
Lest It be done incorrectly or woefully worse
By those not versed the the ways of the Dogcow.

Who's Got the Map?

The ensuing discussion on how to construct an `A5` world is geared primarily to programmers using MPW. There are a couple of reasons for this. First, looking back, the stated problem originated with an error generated by the MPW linker. Other development systems may handle this situation differently and often offer different solutions. Symantec products, for instance, offer `A4`-relative globals and avoid the globals conflict from the outset. Secondly, this document would resemble a Russian novel if it addressed all the permutations of potential solutions for each development system. MPW Pascal is the *de facto* standard Macintosh programming environment for illustrative and educational purposes. It may not be fair, but at least it's consistent.

As already described, there are basically three reasons why stand-alone code might need to reserve space for its own global variables. Consider the following three scenarios as a basis, but understand that various arbitrary combinations are also possible:

- A stand-alone module consists of two functions. There is one main entry point and one function calls another function in the process of calculating its final result. Instead of passing a formal parameter to the subordinate function, the programmer chooses to pass a global.

- A stand-alone module consists of one function. The module is loaded into memory once and invoked multiple times by the host application. The module requires its own private storage to persist across multiple invocations.

- A complex 'INIT' uses QuickDraw, or a 'cdev' is complex enough to require an application-like set of globals to accomplish its self-contained task. A module may need to access data in a Toolbox callback (like a dialog hook) where the interface is fixed, for instance.

Each of the demonstration units is a working example. There is source code at the end of the discussion for simple applications which can play host to these modules and demonstrate how a complete product fits together.

The first instance is relatively easy to implement. When the module is executed, it creates an A5 world, does its job, and then tears down the A5 world, making sure to restore the host application's world. Such a module may look something like the following example. Pay special attention to the items in boldface. These are specific to the use of globals in stand-alone code.

LazyPass.p

```
UNIT LazyPass;

{ This is a stand-alone module which implements the function }
{ of determining a circle's area from its circumference.     }

INTERFACE

  USES
     Types, SAGlobals;

  FUNCTION CircleArea (circumference: Real) : Real;

IMPLEMENTATION

  { Define a variable global to all }
  { of the routines in this unit.   }
   VAR radius : Real;

  FUNCTION RadiusSquared : Real;
  FORWARD;

  { CircleArea is defined first so that the entry point is }
  { conveniently located at the beginning of the module.   }

  FUNCTION CircleArea (circumference: Real) : Real;
  VAR
     A5Ref:  A5RefType;
     oldA5:  Longint;
  BEGIN
     oldA5  :=  OpenA5World(A5Ref);
      radius := circumference / (2.0 * Pi);
      CircleArea := Pi * RadiusSquared;
     CloseA5World(oldA5,  A5Ref);
  END;

  FUNCTION RadiusSquared : Real;
  BEGIN
    RadiusSquared := radius * radius;
  END;

END.
```

LazyPass.make

This is the makefile for the `LazyPass` module.

```
#   File:       LazyPass.make
#   Target:     LazyPass
#   Sources:    LazyPass.p

OBJECTS = LazyPass.p.o

LazyPass ƒƒ LazyPass.make {OBJECTS}
  Link -w -t '????' -c '????' -rt CUST=128 -m CIRCLEAREA -sg LazyPass ∂
    {OBJECTS} ∂
    "{Libraries}"Runtime.o ∂
    "{Libraries}"Interface.o ∂
    "{PLibraries}"SANELib.o ∂
    "{PLibraries}"PasLib.o ∂
    "{MyLibraries}"SAGlobals.o ∂
    -o LazyPass
LazyPass.p.o ƒ LazyPass.make LazyPass.p
   Pascal -i "{MyInterfaces}" LazyPass.p
```

The second instance is a little trickier and requires the cooperation of the host application. The module needs the ability to pass a reference to its global variable storage (`A5` world) back to the application so that it can be easily restored the next time the module is invoked. In addition, there must be some way to notify the module the first time and the last time it is to be called. This kind of module is exemplified by the following:

Persist.p

```
UNIT Persist;

{ This is a stand-alone module which maintains a running total }
{ of the squares of the parameters it receives. This requires  }
{ the cooperation of a host application. The host must use     }
{ messages to tell the module when to initialize and when to   }
{ tear down. The host also must maintain a handle to the       }
{ module's A5 world between invocations.                       }

INTERFACE

  USES
     Types, SAGlobals;

  CONST
    kAccumulate = 0;  {These are the control messages.}
    kFirstTime = 1;
    kLastTime = 2;

  FUNCTION AccSquares (parm: Longint; message: Integer;
        VAR A5Ref: A5RefType) : Longint;

IMPLEMENTATION
  { Define global storage to retain a running }
  { total over multiple calls to the module.  }
   VAR  accumulation  :  Longint;
   FUNCTION AccSquares (parm: Longint; message: Integer;
         VAR A5Ref: A5RefType) : Longint;
  VAR
     oldA5: Longint;
  BEGIN
     IF message = kFirstTime THEN MakeA5World(A5Ref);
     oldA5 := SetA5World(A5Ref);
      IF message = kFirstTime THEN accumulation := 0;
      accumulation := accumulation + (parm * parm);
      AccSquares := accumulation;
     RestoreA5World(oldA5, A5Ref);
     IF message = kLastTime THEN DisposeA5World(A5Ref);
  END;
END.
```

Persist.make

This is the makefile for the `Persist` module.

```
#   File:       Persist.make
#   Target:     Persist
#   Sources:    Persist.p

OBJECTS = Persist.p.o

Persist ƒƒ Persist.make {OBJECTS}
  Link -w -t '????' -c '????' -rt CUST=129 -m ACCSQUARES -sg Persist ∂
    {OBJECTS} ∂
    "{Libraries}"Runtime.o ∂
    "{Libraries}"Interface.o ∂
    "{PLibraries}"SANELib.o ∂
    "{PLibraries}"PasLib.o ∂
    "{MyLibraries}"SAGlobals.o ∂
    -o Persist
Persist.p.o ƒ Persist.make Persist.p
   Pascal -i "{MyInterfaces}" Persist.p
```

BigBro; FORWARD;

The third case is illustrated by an `'INIT'` using arbitrary Toolbox managers to present a user interface. A working example is too long to present here, but an example is included at the end of the discussion. The process, however, is no more difficult than the examples previously given; there is simply more intervening code to accomplish an interesting task. An `'INIT'` may simply call `OpenA5World` upon entry and `CloseA5World` before exiting. Everything between can then be just like an application: `_InitGraf`, `_InitWindows`, and so on. An `'INIT'` should be careful, though, to restore the `GrafPort` to its initial value before exiting.

Dashing Aside the Curtain, or Revealing the Wizard

Building an `A5` world would seem to be fairly complicated, but most of the necessary code is already written. Much of it is in the MPW library called Runtime.o. Actually, this makes sense, since applications have `A5` worlds and the programmer doesn't have to do anything special to set them up. Only in the case of stand-alone modules does this become the responsibility of the programmer. What's not in the MPW library is the initial allocation of space for an `A5` world. For an application, this is done by the Segment Loader. A stand-alone module can emulate the entire process by using bit of glue code around calls to the appropriate routines in Runtime.o. This is the entire point of the `SAGlobals` unit. `SAGlobals` makes it very easy to use globals in stand-alone code because it automates the process of allocating space for globals and initializes them the same way an application would. The Pascal source code for `SAGlobals` is published here. DTS can also provide the source code in C, as well as simplified Pascal and C headers and the compiled object library.

```
{ Stand-alone code modules which need to use global variables
  may include the interfaces in this unit. Such code modules
  must also be linked with Runtime.o and SAGlobals.o. }

UNIT SAGlobals;

INTERFACE

  USES
    Types, Memory, OSUtils;

  TYPE
    A5RefType = Handle;

  { MakeA5World allocates space for an A5 world based on the
    size of the global variables defined by the module and its
    units. If sufficient space is not available, MakeA5World
    returns NIL for A5Ref and further initialization is aborted. }
  PROCEDURE MakeA5World (VAR A5Ref: A5RefType);

  { SetA5World locks down a previously-allocated handle containing
    an A5 world and sets the A5 register appropriately. The return
    value is the old value of A5 and the client should save it for
    use by RestoreA5World. }
  FUNCTION SetA5World (A5Ref: A5RefType) : Longint;

  { RestoreA5World restores A5 to its original value (which the
    client should have saved) and unlocks the A5 world to avoid
    heap fragmentation in cases where the world is used again. }
  PROCEDURE RestoreA5World (oldA5: Longint; A5Ref: A5RefType);

  { DisposeA5World simply disposes of the A5 world handle. }
  PROCEDURE DisposeA5World (A5Ref: A5RefType);

  { OpenA5World combines MakeA5World and SetA5World for the majority
    of cases in which these two routines are called consecutively. An
    exception is when a single A5 world is invoked many times. In this
    case, the world is only created once with MakeA5World and it is
    invoked each time by SetA5World. Most developers will find it easier
    just to call OpenA5World and CloseA5World at the end. If the memory
    allocation request fails, OpenA5World returns NIL for A5Ref and zero
    in the function result. }
  FUNCTION OpenA5World (VAR A5Ref: A5RefType) : Longint;

  { CloseA5World is the dual of OpenA5World. It combines RestoreA5World
    and DisposeA5World. Again, in certain cases it may be necessary to
    call those two routines explicitly, but most of the time CloseA5World
    is all that is required. }
  PROCEDURE CloseA5World (oldA5: Longint; A5Ref: A5RefType);

IMPLEMENTATION

  CONST
    kAppParmsSize = 32;

  FUNCTION A5Size : Longint;
    C;  EXTERNAL;    { in MPW's Runtime.o }

  PROCEDURE A5Init (myA5: Ptr);
    C;  EXTERNAL;    { in MPW's Runtime.o }
```

```
      PROCEDURE MakeA5World (VAR A5Ref: A5RefType);
      BEGIN
        A5Ref := NewHandle(A5Size);
        { The calling routine must check A5Ref for NIL! }
        IF A5Ref <> NIL THEN
          BEGIN
            HLock(A5Ref);
            A5Init(Ptr(Longint(A5Ref^) + A5Size - kAppParmsSize));
            HUnlock(A5Ref);
          END;
      END;

      FUNCTION SetA5World (A5Ref: A5RefType) : Longint;
      BEGIN
        HLock(A5Ref);
        SetA5World := SetA5(Longint(A5Ref^) + A5Size - kAppParmsSize);
      END;

      PROCEDURE RestoreA5World (oldA5: Longint; A5Ref: A5RefType);
      BEGIN
        IF Boolean (SetA5(oldA5)) THEN;     { side effect only }
        HUnlock(A5Ref);
      END;

      PROCEDURE DisposeA5World (A5Ref: A5RefType);
      BEGIN
        DisposHandle(A5Ref);
      END;

      FUNCTION OpenA5World (VAR A5Ref: A5RefType) : Longint;
      BEGIN
        MakeA5World(A5Ref);
        IF A5Ref <> NIL THEN
          OpenA5World := SetA5World(A5Ref)
        ELSE
          OpenA5World := 0;
      END;

      PROCEDURE CloseA5World (oldA5: Longint; A5Ref: A5RefType);
      BEGIN
        RestoreA5World(oldA5, A5Ref);
        DisposeA5World(A5Ref);
      END;

    END.
```

It is tempting to reduce the entire globals issue to this cookbook recipe. The preceding examples may tend to reinforce this view, but a solid theoretical understanding may be indispensable depending on what sort of code goes between `MakeA5World` and `DisposeA5World`. In the Sorter example at the end of this discussion, for instance, an `'XCMD'` makes callbacks to HyperCard. There is a similar mechanism between Apple File Exchange and custom translators. When making these callbacks, it is necessary to temporarily restore the host's `A5` world. Otherwise, the host application bombs when it finds a different set of variables referenced by `A5`. Calling `SetA5` before and after a callback solves the problem, but neither the problem nor the solution is exactly part of the `SAGlobals` recipe. Hence, if a programmer chooses to use the `SAGlobals` unit without understanding how and why it works, he most likely gets in a lot of trouble and ends up writing to Apple to ask why it doesn't work right. As the best mathematics and physics students generally attest: don't just memorize formulas—*know the concepts behind them.*

`A5Size` and `A5Init` are the MPW library routines necessary to set up and initialize an `A5` world. `A5Size` determines how much memory is required for the `A5` world. This memory consists of two parts: memory for globals and memory for application parameters. `A5Init` takes a pointer to the `A5` globals and initializes them to the appropriate values. How this works needs a little explaining.

When MPW links an application together, it has to describe what the globals area should look like. At the very least, it needs to keep track of how large the globals section should be. In addition, it may need to specify what values to put into the globals area. Normally, this means setting everything to zero, but some languages like C allow specification of preinitialized globals. The linker normally creates a packed data block that describes all of this and places it into a segment called `%A5Init`. Also included in this segment are the routines called by the MPW run-time initialization package to act upon this data. `A5Size` and `A5Init` are two such routines. `A5Size` looks at the field that holds the unpacked size of the data and returns it to the caller. `A5Init` is responsible for unpacking the data into the globals section. In the case of a stand-alone module, all code and data needs to be packed into a single segment or resource, so `%A5Init` is not used. The linker option `-sg` is used to make sure that everything is in the same resource. The MPW Commando interface to CreateMake is very good about specifying this automatically, but the programmer must remember to specify this if he creates his own makefiles.

The rest of the `SAGlobals` unit is mostly self-explanatory. The Memory Manager calls straightforwardly allocate the amount of space indicated by `A5Size`, and lock the handle down when in use by the module. If the math performed by `MakeA5World` and `SetA5World` seems just a little too cosmic in nature, don't be alarmed. It's really quite simple. Referring back to Figure 2, `A5` needs to point to the boundary between the global variables and the application parameters. Since the application parameters, including the pointer to QuickDraw globals, are 32 bytes long, the formula should become clear. It's just `starting address + block length - 32`.

As demonstrated in the examples, a module can simply call `MakeA5World` to begin building its own `A5` world, and it can call `SetA5World` to invoke it and make it active. What is not demonstrated particularly well in the examples is that the module should check `A5Ref` to see if it is `NIL`. If so, there is not space to allocate the `A5` world, and the module needs to abort gracefully or find another way of getting its job done. Also, the programmer should be aware that `A5Ref` is **not** an actual `A5` value. It is a reference to an `A5` world as its name implies. The actual value of `A5` is calculated whenever that world is invoked, as described in the preceding paragraph.

Are We There Yet?

As the preceding sections indicate, stand-alone code is one of the more esoteric areas of Macintosh programming. Many more pages could be devoted to the subject, and they probably will be eventually, but there should be enough information here to get most developers past the initial hurdles of creating stand-alone modules and interfacing with an environment biased toward full-blown applications. As always, suggestions for additional topics are welcome and will be incorporated as demand requires and resources permit.

Party on, Dudes.

LazyTest

LazyTest.p

This is a very simple program to demonstrate use of the `LazyPass` module documented earlier. Things to watch out for are standard I/O (`ReadLn` and `WriteLn`) and error checking (or lack thereof). This is a bare-bones example of how to load and call a stand-alone module. Don't expect anything more.

```
PROGRAM LazyTest;
USES
  Types, Resources, Memory, OSUtils;

VAR
  a, c: Real;
  h1: Handle;

  FUNCTION CallModule (parm: Real; modHandle: Handle) : Real;
  INLINE $205F,    { MOVEA.L  (A7)+,A0         pop handle off stack        }
         $2050,    { MOVEA.L  (A0),A0   dereference to get address of XCMD }
         $4E90;    { JSR      (A0)        call XCMD, leaving pb on stack   }

BEGIN
  Write('Circumference:');
  ReadLn(c);

  h1 := GetResource('CUST',128);
  HLock(h1);
  a := CallModule(c,h1);
  HUnlock(h1);

  WriteLn('Area: ',a);
END.
```

LazyTest.make

The accompanying makefile is pretty basic, the kind of thing one expects from CreateMake. The only notable addition is a directive to include the `LazyPass` module in the final application. This avoids the need to paste `LazyPass` into the application manually with ResEdit. It is also an example of a very powerful feature of the MPW scripting language, which allows the output of one command to be "piped" into the input of another.

```
#   File:       LazyTest.make
#   Target:     LazyTest
#   Sources:    LazyTest.p

OBJECTS = LazyTest.p.o

LazyTest ƒƒ LazyTest.make LazyPass
  Echo 'Include "LazyPass";' | Rez -o LazyTest

LazyTest ƒƒ LazyTest.make {OBJECTS}
  Link -w -t APPL -c '????' ∂
    {OBJECTS} ∂
    "{Libraries}"Runtime.o ∂
    "{Libraries}"Interface.o ∂
    "{PLibraries}"SANELib.o ∂
    "{PLibraries}"PasLib.o ∂
    -o LazyTest
LazyTest.p.o ƒ LazyTest.make LazyTest.p
   Pascal  LazyTest.p
```

PersistTest

PersistTest.p

PersistTest is an equally minimal application to demonstrate the Persist module, also documented earlier.

```
PROGRAM PersistTest;

USES
  Types, Resources, Memory, OSUtils;

CONST
  N = 5;
  kAccumulate = 0;       {These are the control messages.}
  kFirstTime = 1;
  kLastTime = 2;

VAR
  i : Integer;
  acc : Longint;
  h1, otherA5: Handle;

  FUNCTION CallModule (parm: Longint; message: Integer; VAR otherA5: Handle;
    modHandle: Handle) : Longint;
  INLINE $205F,    { MOVEA.L  (A7)+,A0         pop handle off stack        }
         $2050,    { MOVEA.L  (A0),A0  dereference to get address of XCMD }
         $4E90;    { JSR      (A0)        call XCMD, leaving pb on stack   }

BEGIN
  h1 := GetResource('CUST',129);
  MoveHHi(h1);
  HLock(h1);

  FOR i := 1 TO N DO
    BEGIN
      CASE i OF
        1: acc := CallModule(i,kFirstTime,otherA5,h1);
        N: acc := CallModule(i,kLastTime,otherA5,h1);
      OTHERWISE
        acc := CallModule(i,kAccumulate,otherA5,h1);
      END;
      WriteLn('SumSquares after ',i,' = ',acc);
    END;
  HUnlock(h1);
END.
```

PersistTest.make

This makefile presents nothing new and is provided for the sake of completeness.

```
#   File:       PersistTest.make
#   Target:     PersistTest
#   Sources:    PersistTest.p

OBJECTS = PersistTest.p.o

PersistTest ƒƒ PersistTest.make Persist
  Echo 'Include "Persist";' | Rez -o PersistTest

PersistTest ƒƒ PersistTest.make {OBJECTS}
  Link -w -t APPL -c '????' ∂
    {OBJECTS} ∂
    "{Libraries}"Runtime.o ∂
    "{Libraries}"Interface.o ∂
    "{PLibraries}"SANELib.o ∂
    "{PLibraries}"PasLib.o ∂
    -o PersistTest
PersistTest.p.o ƒ PersistTest.make PersistTest.p
   Pascal  PersistTest.p
```

Sorter

Sorter.p

`Sorter` is an example `'XCMD'` which demonstrates the concept of persistent globals across multiple invocations. It also illustrates how stand-alone modules must handle callbacks to a host application. This is evidenced by the `SetA5` instructions bracketing HyperCard callback routines, such as `ZeroToPas`, `SetGlobal`, or user routines incorporating such calls.

```
{$Z+} { This allows the Linker to find "ENTRYPOINT" without our having to put it
       in the INTERFACE section }

UNIT Fred;

  INTERFACE

    USES
      Types, Memory, OSUtils, HyperXCmd, SAGlobals;

  IMPLEMENTATION

    TYPE
      LongArray = ARRAY [0..0] OF Longint; { These define our list of entries }
      LongPointer = ^LongArray;
      LongHandle = ^LongPointer;

    CONST
      kFirstTime = 1;     { being called for the first time. Initialize. }
      kLastTime = 2;      { being called for the last time. Clean up. }
      kAddEntry = 3;      { being called to add an entry to our list to sort. }
      kSortEntries = 4;   { being called to sort and display our list. }
  kCommandIndex = 1;  { Parameter 1 holds our command number. }
  kA5RefIndex = 2;    { Parameter 2 holds our A5 world reference. }
  kEntryIndex = 3;    { Parameter 3 holds a number to add to our list. }

VAR
  gHostA5: Longint; { The saved value of our host's (HyperCard's) A5. }
  gNumOfEntries: Longint; { The number of entries in our list. }
  gEntries: LongHandle; { Our list of entries. Gets expanded as needed. }

  { Forward reference to the main procedure. This is so we can jump to
  it from ENTRYPOINT, which represents the beginning of the XCMD, and is
  what HyperCard calls when it calls us. }

PROCEDURE Sorter(paramPtr: XCmdPtr);
  FORWARD;

PROCEDURE ENTRYPOINT(paramPtr: XCmdPtr);

  BEGIN
    Sorter(paramPtr);
  END;

{ Utility routines for using the HyperCard callbacks. There are some
  functions that we need to perform many times, or would like to
  encapsulate into little routines for clarity:

    ValueOfExpression - given an index from 1 to 16, this evaluates the
        expression of that parameter. This is used to scoop out the value
        of the command selector, our A5 pointer, and the value of the
        number we are to stick into our list of numbers to sort.

    LongToZero - Convert a LONGINT into a C (zero delimited) string.
        Returns a handle that contains that string.

    SetGlobalAt - given the index to one of the 16 parameters and a
        LONGINT, this routines sets the global found in that parameter to
        the LONGINT.
}

FUNCTION ValueOfExpression(paramPtr: XCmdPtr;
                           index: integer): Longint;

  VAR
    tempStr: Str255;
    tempHandle: Handle;

  BEGIN
    ZeroToPas(paramPtr, paramPtr^.params[index]^, tempStr);
    tempHandle := EvalExpr(paramPtr, tempStr);
    ZeroToPas(paramPtr, tempHandle^, tempStr);
    DisposHandle(tempHandle);
    ValueOfExpression := StrToLong(paramPtr, tempStr);
  END;

FUNCTION LongToZero(paramPtr: XCmdPtr;
                    long: Longint): Handle;

  VAR
    tempStr: Str255;

  BEGIN
    LongToStr(paramPtr, long, tempStr);
    LongToZero := PasToZero(paramPtr, tempStr);
  END;
```

```
PROCEDURE SetGlobalAt(paramPtr: XCmdPtr;
                      index: integer;
                      long: Longint);

  VAR
    globalName: Str255;
    hLong: Handle;

  BEGIN
    ZeroToPas(paramPtr, paramPtr^.params[index]^, globalName);
    hLong := LongToZero(paramPtr, long);
    SetGlobal(paramPtr, globalName, hLong);
    DisposHandle(hLong);
  END;

{ These 4 routines are called according to the command passed to the XCMD:

    Initialize - used to initialize our globals area. A5Init will clear
        our globals to zero, and set up any pre-initialized variables if we
        wrote our program in C or Assembly, but it can't do everything. For
        instance, in this XCMD, we need to create a handle to hold our list
        of entries.
    AddAnEntry - Takes the value represented by the 3 parameters passed to
        us by HyperCard and adds it to our list.
    SortEntries - Sorts the entries we have so far. Converts them into a
        string and tells HyperCard to display them in the message box.
    FreeData - We just receive the message saying that we are never going
        to be called again. Therefore, we must get rid of any memory we
        have allocated for our own use.
}

PROCEDURE Initialize;

  BEGIN
    gEntries := LongHandle(NewHandle(0));
    gNumOfEntries := 0;
  END;

PROCEDURE AddAnEntry(paramPtr: XCmdPtr);

  VAR
    ourA5: Longint;
    tempStr: Str255;
    temp: Longint;

  BEGIN
    ourA5 := SetA5(gHostA5);
    temp := ValueOfExpression(paramPtr, kEntryIndex);
    ourA5 := SetA5(ourA5);

    SetHandleSize(Handle(gEntries), (gNumOfEntries + 1) * 4);
    {$PUSH} {$R-}
    gEntries^^[gNumOfEntries] := temp;
    {$POP}
    gNumOfEntries := gNumOfEntries + 1;
  END;

PROCEDURE SortEntries(paramPtr: XCmdPtr);

  VAR
    ourA5: Longint;
    i, j: integer;
    fullStr: Str255;
    tempStr: Str255;
    temp: Longint;
  BEGIN
    IF gNumOfEntries > 1 THEN
      BEGIN
      {$PUSH} {$R-}
      FOR i := 0 TO gNumOfEntries - 2 DO
        BEGIN
        FOR j := i + 1 TO gNumOfEntries - 1 DO
          BEGIN
          IF gEntries^^[i] > gEntries^^[j] THEN
            BEGIN
            temp := gEntries^^[i];
            gEntries^^[i] := gEntries^^[j];
            gEntries^^[j] := temp;
            END;
          END;
        END;
      {$POP}
      END;

    IF gNumOfEntries > 0 THEN
      BEGIN
      fullStr := '';
      FOR i := 0 TO gNumOfEntries - 1 DO
        BEGIN
        {$PUSH} {$R-}
        temp := gEntries^^[i];
        {$POP}
        ourA5 := SetA5(gHostA5);
        NumToStr(paramPtr, temp, tempStr);
        ourA5 := SetA5(ourA5);
        fullStr := concat(fullStr, ', ', tempStr);
        END;
      delete(fullStr, 1, 2); { remove the first ", " }
      ourA5 := SetA5(gHostA5);
      SendHCMessage(paramPtr, concat('put "', fullStr, '"'));
      ourA5 := SetA5(ourA5);
      END;
  END;

PROCEDURE FreeData;

  BEGIN
    DisposHandle(Handle(gEntries));
  END;

{ Main routine. Big Cheese. Head Honcho. The Boss. The Man with all the
  moves. You get the idea. This is the controlling routine. It first
  checks to see if we have the correct number of parameters (sort of).
  If that's OK, then it either creates a new A5 world and initializes it,
  or it sets up one that we've previously created.  It then dispatches to
  the appropriate routine, depending on what command was passed to us.
  Finally, it restores the host application's A5 world, and disposes of
  ours if this is the last time we are being called. }

PROCEDURE Sorter(paramPtr: XCmdPtr);

  VAR
    command: integer;
    A5Ref: A5RefType;
    errStr: Str255;
    A5Name: Str255;
```

```
      BEGIN {Main}

        WITH paramPtr^ DO
          IF (paramCount < 2) OR (paramCount > 3) THEN
            BEGIN
            errStr := 'Correct usage is: "Sorter <function> <A5> [<entry>]"';
            paramPtr^.returnValue := PasToZero(paramPtr, errStr);
            EXIT(Sorter); {leave the XCMD}
            END;

        command := ValueOfExpression(paramPtr, kCommandIndex);

        IF command = kFirstTime THEN
          BEGIN
          MakeA5World(A5Ref);
          SetGlobalAt(paramPtr, kA5RefIndex, Longint(A5Ref));
          END
        ELSE
          BEGIN
          A5Ref := A5RefType(ValueOfExpression(paramPtr, kA5RefIndex));
          END;

        IF (A5Ref = NIL) THEN
          BEGIN
          errStr := 'Could not get an A5 World!!!';
          paramPtr^.returnValue := PasToZero(paramPtr, errStr);
          EXIT(Sorter); {leave the XCMD}
          END;

        gHostA5 := SetA5World(A5Ref);
        CASE command OF
          kFirstTime: Initialize;
          kAddEntry: AddAnEntry(paramPtr);
          kSortEntries: SortEntries(paramPtr);
          kLastTime: FreeData;
        END;
        RestoreA5World(gHostA5, A5Ref);

        IF command = kLastTime THEN DisposeA5World(A5Ref)

      END; {main}

END.
```

Sorter.make

The makefile for Sorter is fairly straightforward, but CreateMake cannot generate all of it automatically. Be sure to link with both HyperXLib.o and SAGlobals.o, and account for any custom directories to search for interfaces. In most of the examples, there are two MPW Shell variables, MyInterfaces and MyLibraries which represent the directories containing the SAGlobals headers and library, respectively. Someone following along with these examples would need to define these Shell variables, possibly in his UserStartup file, or replace the occurrences with the name of whatever directory actually contains the necessary SAGlobals files.

```
#   File:       Sorter.make
#   Target:     Sorter
#   Sources:    Sorter.p
```

```
OBJECTS = Sorter.p.o

Sorter ƒƒ Sorter.make {OBJECTS}
  Link -w -t '????' -c '????' -rt XCMD=256 -m ENTRYPOINT -sg Sorter ∂
    {OBJECTS} ∂
    "{Libraries}"Runtime.o ∂
    "{Libraries}"Interface.o ∂
    "{PLibraries}"SANELib.o ∂
    "{PLibraries}"PasLib.o ∂
    "{Libraries}"HyperXLib.o ∂
    "{MyLibraries}"SAGlobals.o ∂
    -o Sorter
Sorter.p.o ƒ Sorter.make Sorter.p
   Pascal -i "{MyInterfaces}" Sorter.p
```

A Sample HyperCard Script Using Sorter

To test Sorter, it is necessary to create a simple HyperCard stack. After creating a new stack under HyperCard's File menu, use the button tool to create a new button and associate it with the following script. Now use ResEdit to paste the 'XCMD' resource "Sorter" into the stack and it's ready for experimentation.

```
on mouseUp
  global A5
  Sorter 1, "A5"        -- Initialize that puppy
  if the result is empty then
    Sorter 3, A5, 6     -- Add some numbers to the list
    Sorter 3, A5, 2
    Sorter 3, A5, 9
    Sorter 3, A5, 12
    Sorter 3, A5, 7
    Sorter 4, A5        -- sort them and print them
    Sorter 2, A5        -- Dispose of our data
  else
    put the result
  end if
end mouseUp
```

BigBro

BigBro.p

BigBro may look a bit familiar because it performs the same function as the sample INIT offered early in the preceding discussion. However, it has the added feature of providing a user interface, or a dialog at least, during the startup sequence. This tends to make it very obnoxious, and DTS discourages this sort of thing on human interface grounds. Nonetheless, it is an interesting case study. It is also the first example in which a stand-alone code resource uses other resources.

```
UNIT BigBro;

INTERFACE

  USES
    Types, SAGlobals, OSUtils,
    QuickDraw, Fonts, Windows, Menus, TextEdit, Dialogs,
    Resources, Sound, ToolUtils;
  PROCEDURE BeAPest;

IMPLEMENTATION

  PROCEDURE BeAPest;
  CONST
    kBigBroDLOG = 128;
  VAR
    A5Ref: A5RefType;
    oldA5: Longint;
    numSnds, i, itemHit: Integer;
    theSnd: Handle;
    playStatus: OSErr;
    orwell: DialogPtr;

  BEGIN
    IF NOT Button THEN BEGIN
      oldA5 := OpenA5World(A5Ref);
      IF A5Ref <> NIL THEN BEGIN
        InitGraf(@thePort);
        InitFonts;
        InitWindows;
        InitMenus;
        TEInit;
        InitDialogs(NIL);
        InitCursor;
        orwell := GetNewDialog(kBigBroDLOG, NIL, WindowPtr(-1));
        numSnds := CountResources('snd ');
        FOR i := 1 TO numSnds DO BEGIN
          theSnd := GetIndResource('snd ',i);
          IF theSnd <> NIL THEN
            playStatus := SndPlay(NIL,theSnd,FALSE);
          END;
        REPEAT
          ModalDialog(NIL, itemHit);
        UNTIL itemHit = 1;
        DisposDialog(orwell);
      CloseA5World(oldA5, A5Ref);
      END;
    END;
  END;

END.
```

BigBro.r

This is the Rez input file necessary to create the 'DLOG' and 'DITL' resources used by `BigBro`.

```
resource 'DLOG' (128) {
  {84, 124, 192, 388},
  dBoxProc,
  visible,
  noGoAway,
  0x0,
  128,
  ""
};
```

```
resource 'DITL' (128) {
  {  /* array DITLarray: 2 elements */
    /* [1] */
    {72, 55, 93, 207},
    Button {
      enabled,
      "Continue Booting"
    },
    /* [2] */
    {13, 30, 63, 237},
    StaticText {
      disabled,
      "This is an exaggerated case of the type "
      "of INIT which bothers me more than anyth"
      "ing else."
    }
  }
};
```

BigBro.make

The makefile for `BigBro` is a little simpler than that of `Sorter`, but includes an extra directive to include the dialog resources using Rez. Refer to the `Sorter` example for notes on the `MyInterfaces` and `MyLibraries` Shell variables.

```
#   File:       BigBro.make
#   Target:     BigBro
#   Sources:    BigBro.p


OBJECTS = BigBro.p.o

BigBro ƒƒ BigBro.make BigBro.r
  Rez -o BigBro "{RIncludes}"Types.r BigBro.r


BigBro ƒƒ BigBro.make {OBJECTS}
  Link -w -t INIT -c '????' -rt INIT=128 -ra =resLocked -m BEAPEST -sg BigBro ∂
    {OBJECTS} ∂
    "{Libraries}"Runtime.o ∂
    "{Libraries}"Interface.o ∂
    "{PLibraries}"SANELib.o ∂
    "{PLibraries}"PasLib.o ∂
    "{MyLibraries}"SAGlobals.o ∂
    -o BigBro
BigBro.p.o ƒ BigBro.make BigBro.p
   Pascal -i "{MyInterfaces}" BigBro.p
```

MyWindowDef

MyWindowDef.a

Writing a 'WDEF' is like writing an 'INIT', except that 'WDEF' resources have standard headers that are incorporated into the code. In this example, the 'WDEF' is the Pascal MyWindowDef. To create the header, use an assembly language stub:

```
StdWDEF    MAIN EXPORT                    ; this will be the entry point
           IMPORT MyWindowDef             ; name of Pascal FUNCTION that is the WDEF
                                          ; we IMPORT externally referenced routines
                                          ; from Pascal (in this case, just this one)
           BRA.S    @0                    ; branch around the header to the actual code
           DC.W     0                     ; flags word
           DC.B     'WDEF'                ; type
           DC.W     3                     ; ID number
           DC.W     0                     ; version
@0         JMP      MyWindowDef           ; this calls the Pascal WDEF
           END
```

MyWindowDef.p

Now for the Pascal source for the 'WDEF'. Only the shell of what needs to be done is listed, the actual code is left as an exercise for the reader (for further information about writing a 'WDEF', see *Inside Macintosh*, Volume I, The Window Manager (pp. 297-302) and Volume V, The Window Manager (pp. 205-206).

```
UNIT WDef;

INTERFACE

  USES MemTypes, QuickDraw, OSIntf, ToolIntf;

{this is the only external routine}
  FUNCTION MyWindowDef(varCode: Integer; theWindow: WindowPtr; message: Integer;
                       param: LongInt): LongInt; {As defined in IM p. I-299}

IMPLEMENTATION

  FUNCTION MyWindowDef(varCode: Integer; theWindow: WindowPtr; message: Integer;
                       param: LongInt): LongInt;

    TYPE
      RectPtr = ^Rect;

    VAR
      aRectPtr : RectPtr;

  {here are the routines that are dispatched to by MyWindowDef}

    PROCEDURE DoDraw(theWind: WindowPtr; DrawParam: LongInt);
      BEGIN {DoDraw}
        {Fill in the code!}
      END; {DoDraw}

    FUNCTION DoHit(theWind: WindowPtr; theParam: LongInt): LongInt;
      BEGIN {DoHit}
        {Code for this FUNCTION goes here}
      END;  {DoHit}
```

```
        PROCEDURE DoCalcRgns(theWind: WindowPtr);
          BEGIN {DoCalcRgns}
            {Code for this PROCEDURE goes here}
          END;  {DoCalcRgns}

        PROCEDURE DoGrow(theWind: WindowPtr; theGrowRect: Rect);
          BEGIN {DoGrow}
            {Code for this PROCEDURE goes here}
          END;  {DoGrow}

        PROCEDURE DoDrawSize(theWind: WindowPtr);
          BEGIN {DoDrawSize}
            {Code for this PROCEDURE goes here}
          END;  {DoDrawSize}

      {now for the main body to MyWindowDef}
      BEGIN  { MyWindowDef }
      {case out on the message and jump to the appropriate routine}
        MyWindowDef := 0; {initialize the function result}

        CASE message OF
          wDraw:  { draw window frame}
            DoDraw(theWindow,param);

          wHit:   { tell what region the mouse was pressed in}
            MyWindowDef := DoHit(theWindow,param);

          wCalcRgns: { calculate structRgn and contRgn}
            DoCalcRgns(theWindow);

          wNew:   { do any additional initialization}
            { we don't need to do any}
            ;

          wDispose:{ do any additional disposal actions}
            { we don't need to do any}
            ;

          wGrow:  { draw window's grow image}
            BEGIN
              aRectPtr := RectPtr(param);
              DoGrow(theWindow,aRectPtr^);
            END; {CASE wGrow}

          wDrawGIcon:{ draw Size box in content region}
            DoDrawSize(theWindow);

        END; {CASE}
      END; {MyWindowDef}
END. {of UNIT}
```

MyWindowDef.make (Pascal Version)

```
#   File:       MyWindowDef.make
#   Target:     MyWindowDef
#   Sources:    MyWindowDef.a MyWindowDef.p

OBJECTS = MyWindowDef.a.o MyWindowDef.p.o

MyWindowDef ƒƒ MyWindowDef.make {OBJECTS}
  Link -w -t '????' -c '????' -rt WDEF=3 -m STDWDEF -sg MyWindowDef ∂
    {OBJECTS} ∂
    -o MyWindowDef
MyWindowDef.a.o ƒ MyWindowDef.make MyWindowDef.a
   Asm  MyWindowDef.a
MyWindowDef.p.o ƒ MyWindowDef.make MyWindowDef.p
   Pascal  MyWindowDef.p
```

That’s all there is to it.

MyWindowDef.c

Writing a `'WDEF'` in MPW C is very similar to writing one in Pascal. You can use the same assembly language header, and all you need to make sure of is that the main dispatch routine (in this case: `MyWindowDef`) is first in your source file. Here’s the same `'WDEF'` shell in MPW C:

```
/* first, the mandatory includes */
#include <types.h>
#include <quickdraw.h>
#include <resources.h>
#include <fonts.h>
#include <windows.h>
#include <menus.h>
#include <textedit.h>
#include <events.h>

/* declarations */
void DoDrawSize();
void DoGrow();
void DoCalcRgns();
long int DoHit();
void DoDraw();

/*----------------------- Main Proc within WDEF ----------------------*/
pascal long int MyWindowDef(varCode,theWindow,message,param)
short int    varCode;
WindowPtr    theWindow;
short int    message;
long int     param;

{    /* MyWindowDef */

  Rect        *aRectPtr;
  long int    theResult=0;       /*this is what the function returns, init to 0 */
  switch (message)
  {
    case wDraw:                  /* draw window frame*/
      DoDraw(theWindow,param);
      break;
    case wHit:                   /* tell what region the mouse was pressed in*/
      theResult = DoHit(theWindow,param);
      break;
    case wCalcRgns:              /* calculate structRgn and contRgn*/
      DoCalcRgns(theWindow);
      break;
    case wNew:                   /* do any additional initialization*/
      break;                     /* nothing here */
    case wDispose:               /* do any additional disposal actions*/
      break;                     /* we don't need to do any*/
    case wGrow:                  /* draw window's grow image*/
      aRectPtr = (Rect *)param;
      DoGrow(theWindow,*aRectPtr);
      break;
    case wDrawGIcon:             /* draw Size box in content region*/
      DoDrawSize(theWindow);
      break;
  } /* switch */
  return theResult;
}    /* MyWindowDef */

/* here are the routines that are dispatched to by MyWindowDef

/*---------------------------- DoDraw function -----------------------------*/
void DoDraw(WindToDraw,DrawParam)
WindowPtr    WindToDraw;
long int     DrawParam;

{  /* DoDraw */
      /* code for DoDraw goes here */
}  /* DoDraw */

/*---------------------------- DoHit function -----------------------------*/
long int DoHit(WindToTest,theParam)
WindowPtr    WindToTest;
long int     theParam;

{  /* DoHit */
      /* code for DoHit goes here */
}  /* DoHit */

/*------------------------- DoCalcRgns procedure --------------------------*/
void DoCalcRgns(WindToCalc)
WindowPtr    WindToCalc;

{  /* DoCalcRgns */
      /* code for DoCalcRgns goes here */
}  /* DoCalcRgns */

/*--------------------------- DoGrow procedure -----------------------------*/
void DoGrow(WindToGrow,theGrowRect)
WindowPtr    WindToGrow;
Rect         theGrowRect;

{  /* DoGrow */
      /* code for DoGrow goes here */
}  /* DoGrow */
```

```
/*------------------------ DoDrawSize procedure --------------------------*/
void DoDrawSize(WindToDraw)
WindowPtr    WindToDraw;

{  /* DoDrawSize */
      /* code for DoDrawSize goes here */
}  /* DoDrawSize */
```

MyWindowDef.make (C Version)

```
#   File:       MyWindowDef.make
#   Target:     MyWindowDef
#   Sources:    MyWindowDef.a MyWindowDef.c

OBJECTS = MyWindowDef.a.o MyWindowDef.c.o

MyWindowDef ƒƒ MyWindowDef.make {OBJECTS}
  Link -w -t '????' -c '????' -rt WDEF=3 -m STDWDEF -sg MyWindowDef ∂
    {OBJECTS} ∂
    -o MyWindowDef
MyWindowDef.a.o ƒ MyWindowDef.make MyWindowDef.a
   Asm  MyWindowDef.a
MyWindowDef.c.o ƒ MyWindowDef.make MyWindowDef.c
   C  -w MyWindowDef.c
```

Debugger 'FKEY'

DebugKey.a

DebugKey a very simple assembly-language example of how to write an 'FKEY' code resource, which traps to the debugger. With this 'FKEY', you can enter the debugger using the keyboard rather than pressing the interrupt switch on your Macintosh.

The build process is a little different for this example, as it links the 'FKEY' directly into the System file. Another script can remove the 'FKEY' resource. If the prospect of turning MPW tools loose on the System file is just too much to bear, the 'FKEY' can be linked into a separate file and pasted into the System file with a more mainstream tool like ResEdit.

```
; File: DebugKey.a
;
; An FKEY to invoke the debugger via command-shift-8
;
DebugKey    MAIN

            BRA.S    CallDB    ;Invoke the debugger
            ;standard header
            DC.W     $0000     ;flags
            DC.L     'FKEY'    ;'FKEY' is 464B4559 hex
            DC.W     $0008     ;FKEY Number
            DC.W     $0000     ;Version number
CallDB      DC.W     $A9FF     ;Debugger trap
            RTS
            END
```

InstallDBFKEY (An MPW Installation Script)

```
#  DebugKey Installer Script
#
#  Place this file in the current directory and type
#  "InstallDBFKEY <Enter>" to install the debugger FKEY
#  in your System file.
#
  Asm DebugKey.a
  Link DebugKey.a.o -o "{SystemFolder}"System -rt FKEY=8
```

Further Reference:

- *Inside Macintosh*, Volumes I & V, The Window Manager
- *Inside Macintosh*, Volume II, The Memory Manager & The Segment Loader
- *Inside Macintosh*, Volume V, The Start Manager
- *MPW Reference Manual*
- Technical Note #208, Setting and Restoring A5
- Technical Note #240, Using MPW for Non-Macintosh 68000 Systems

Macintosh Technical Notes

Developer Technical Support

#257: Slot Interrupt Prio-Technics

Written by: Mark Baumwell October 1989

This Technical Note describes the way interrupt priorities are scheduled, which corrects the description of slot interrupt queue priorities in the Device Manager chapter of *Inside Macintosh*, Volume V-426.

According to *Inside Macintosh*, Volume V-426, The Device Manager, the `SQPrio` field of a slot interrupt queue element is an unsigned byte that determines the order in which slots are polled and interrupt service routines are called. This is **incorrect** on all Macintosh models prior to the IIci that are running a system version earlier than System Software 7.0.

In reality, slot interrupts of **lower** priority values have always been called first. However, all new Macintosh computers, starting with the Macintosh IIci, as well as all machines running System Software 7.0 or later, will have an `_SIntInstall` routine that has been changed to reflect the description in *Inside Macintosh*.

In addition, the `SQPrio` field is, and has always been, two bytes long, but the high byte is reserved and must be set to zero.

Apple still reserves priority values 200-255 as documented in *Inside Macintosh*.

Note that in any case of slot interrupts with equal priority, the most recently installed interrupt is run first, regardless of system version.

Further Reference:

- *Inside Macintosh*, Volume V-426, The Device Manager

Macintosh
Technical Notes

Developer Technical Support

#258: Our Checksum Bounced

Written by: Jim Reekes October 1989

This Technical Note discusses a fix to a SCSI Manager bug which concerns all developers working with SCSI and NuBus™ device drivers.

A Bit of History

The boot code contained in the ROM has a feature used by the Start Manager to perform a checksum on the SCSI driver being loaded. *Inside Macintosh*, Volume V-573, The SCSI Manager, documents this being performed on the Macintosh SE and later models for volumes using the new partitioning method. The truth, however, is that that checksum verification was never performed due to a bug in the ROM, and because of this, all drivers loaded regardless of validity.

That was the case until recently. On new Macintosh computers, the checksum verification works. That's the good news: we've fixed the bug. Now the bad news: this fix causes a number of third-party SCSI drivers to fail to load.

Some SCSI drivers improperly implement the new partitioning scheme. If the partition map entry name begins with the four letters "Maci" (case sensitive) and is of type "Apple_Driver", the driver now has its checksum verified with the entries in the partition map. If this checksum fails, the driver is not loaded. This checksum algorithm is documented in *Inside Macintosh*, Volume V-573, The SCSI Manager.

Drivers That Check In, But Don't Check Out

The checksum routine tests the number of bytes specified in `pmBootSize`, beginning at the start of the driver boot code. Only drivers contained within the new partition map have this test performed. If you are using the old partition map scheme documented in *Inside Macintosh*, Volume IV-283, The SCSI Manager, the driver does not have its checksum validated. The following is the startup logic in the new Macintosh ROMs:

```
IF
        pmSig = $504D
AND
        pmPartName = Maci
AND
        pmPartType = Apple_Driver
AND
        pmBootChecksum = ChecksumOf(bootCode, pmBootSize)
THEN
        Load the driver
ELSE
        Do not load the driver
```

Just When You Thought It Was Safe To Call _SysEnvirons

The call `_SysEnvirons` was created for compatibility reasons. It allows an application to make a single call to the system to determine its characteristics. It keeps the application from reading ROM addresses and low memory. This trap is now in the ROM of new machines. But, before you get excited about this addition to ROM, there is something that *Inside Macintosh*, Volume V-5, Compatibility Guidelines, states that must be understood by those writing SCSI drivers:

> *"All of the Toolbox Managers must be initialized before calling SysEnvirons."…"SysEnvirons is not intended for use by device drivers, but can be called from desk accessories."*

This statement means that neither SCSI nor NuBus device drivers can use `_SysEnvirons`. The earliest possible moment to call `_SysEnvirons` is at INIT time. Some SCSI drivers call `_SysEnvirons`, and this causes the Macintosh to crash at boot time.

To Sum Up

Check if your partition map is of the version described in the SCSI Manager chapter of *Inside Macintosh*, Volume V, and contains the `pmPartName` and `pmPartType` as mentioned earlier in this Note. If it does, then verify that the `pmBootChecksum` is correct. If the checksum is not correct, the new Macintosh computers will not load your driver.

The solution to this problem is to have a valid partition map entry in all cases and to expect the Start Manager to perform the checksum verification regardless of the `machineType`. `_SysEnvirons` is not available until the system has been initialized.

Further Reference:

- *Inside Macintosh*, Volume IV-283, The SCSI Manager
- *Inside Macintosh*, Volume V-5, Compatibility Guidelines
- *Inside Macintosh*, Volume V-573, The SCSI Manager
- Technical Note #129, _SysEnvirons: System 6.0 and Beyond

NuBus is a trademark of Texas Instruments

#259: Old Style Colors

Revised by: Rich "I See Colors" Collyer August 1990
Written by: Rich "I See Colors" Collyer & Byron Han October 1989

This Technical Note covers limitations of the original Macintosh color model (eight-color) which *Inside Macintosh*, Volume I-173, QuickDraw does not document.
Changes since October 1989: Added definitions of the old-style constants.

QuickDraw has always been able to deal with color, just on a very limited basis. Most applications have not made use of this feature, since Color QuickDraw-based Macintoshes come with a better color model. There are, however, a few nice features which come with the old style color model. With the old style colors, it is easy to print color on an ImageWriter with a color ribbon. Another advantage is that developers do not have to write special-case code depending upon whether or not a machine has Color QuickDraw.

Now that you are ready to convert to the old style colors, there are a few things you should know about which do not work with old style colors. This Note covers the limitations of using old style colors, as well as the best ways to work around these limitations.

Limitations

The most obvious limitation is that of only eight colors: black, white, red, green, blue, cyan, yellow, and magenta. This limitation is only a problem if you want to produce a color-intensive application; if this describes your application, then you need not read any further in this Note.

The next limitation is that off-screen buffers are not very useful. You can draw into off-screen buffers, but there is no way to get the colors back from the buffer. This leads into the next limitation, which is that `_CopyBits` cannot copy more than one color at a time.

When you call `_CopyBits` from an off-screen buffer to your window, you need to set the forecolor to the color you want to copy before calling `_CopyBits` (i.e., to copy a red object, call `_ForeColor(redColor)`). Now when you copy the object, you can only copy one color. If you copy different colored objects at one time, then you have a problem. The result of a multicolored copy is that all objects copy in the same color, that of the foreground.

It is possible to work with an off-screen buffer and the old style colors, but it requires a lot of extra work. Unless the objects are really complex, then it is probably easier to just draw the objects directly into your window.

One other limitation does exist. Consider the following code sample. One would assume that this sample would work at all times.

```
SetPort (myPort);
savedFG := myPort^.fgColor;
ForeColor (redColor);          {or any other color}

{...drawing takes place here...}

ForeColor (savedFG);
```

Surprise. It does not always work. The saved value for the `fgColor` field of the `GrafPort` is not a classic QuickDraw color if the `GrafPort` is actually a `CGrafPort`. If dealing with a `CGrafPort`, the `fgColor` field actually contains the foreground color's entry in the color table, so the second call to `_ForeColor` really messes things up.

The proper way to set and reset the foreground color with classic QuickDraw's `_ForeColor` call is as follows:

```
SetPort (myPort);
savedFG := myPort^.fgColor;
ForeColor (redColor);          {or any other color}

{...drawing takes place here...}

myPort^.fgColor := savedFG;    {manually stuff the old fgColor back}
If (32BQD = TRUE) Then         {32BQD is a flag which is made and set by}
    PortChanged (myPort);      {the application; to set it, the application}
                               {needs to check _Gestalt for 32-Bit QuickDraw}
```

This Note also applies to the routine `_BackColor`.

What Works

The easiest way to work with these limited colors is to use pictures. When you draw the images, you should draw into a picture. Then when you want to draw the images into your window or to a printer, call `_DrawPicture`. Pictures work well with the old style colors, and you don't need to worry about making sure that the forecolor is current when you draw into your window.

Once you have the picture, you can use it to draw into the screen or to the printer port. You can also set the `WindowRecords` `windowPic` to equal your `PictureHandle` so updates are handled by the Window Manager.

What Do Those Constants Mean Anyway

Each of the constants contains nine bits of information, and each bit has a special meaning. Figure 1 illustrates the meaning of each of the bits, while Table 1 shows how each of the color constants fills in the appropriate bits.

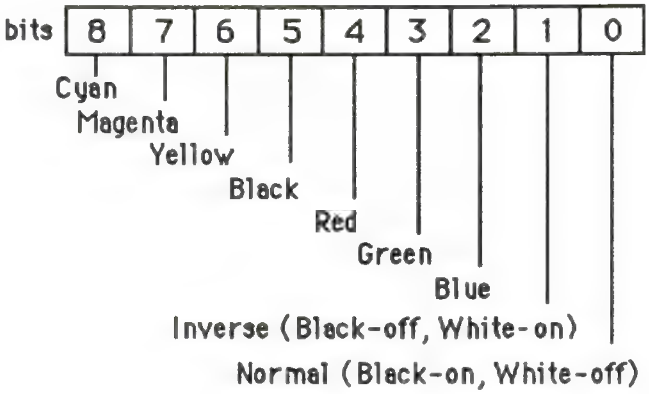


Figure 1–Bit Definitions

	black	white	red	green	blue	cyan	magenta	yellow
	(33)	(30)	(209)	(329)	(389)	(269)	(149)	(89)
Cyan	0	0	0	1	1	1	0	0
Magenta	0	0	1	0	1	0	1	0
Yellow	0	0	1	1	0	0	0	1
Black	1	0	0	0	0	0	0	0
Red	0	1	1	0	0	0	1	1
Green	0	1	0	1	0	1	0	1
Blue	0	1	0	0	1	1	1	0
Inverse	0	1	0	0	0	0	0	0
Normal	1	0	1	1	1	1	1	1

Table 1–Color-Bit Correlation

Further Reference:

- *Inside Macintosh*, Volume I-173, QuickDraw
- Technical Note #277, Of Time and Space and _CopyBits

Macintosh
Technical Notes

Developer Technical Support

#260: NuBus Power Allocation

Written by: Rich “I See Colors” Collyer October 1989

This Technical Note discusses a very real power limit for NuBus™ expansion cards and warns developers to heed this limit lest they want users trashing their machines by overextending the Macintosh power supply.

Click-Click Mode?

Designing Cards and Drivers for the Macintosh clearly states that allowed power per NuBus slot is 13.9 watts (pg. 6-6). That is 2 amps at 5 volts, 0.175 amps at 12 volts, and 0.15 amps at -12 volts. If your Nubus card requires more than this allocation, then you need to make sure that users do not fill all of their Nubus slots. A good rule of thumb is that if users can fill all of their slots with your card and the machine is still able to boot, then you are okay. If the machine goes into click-click mode, then you need to make sure that users cannot fill their slots. Click-click mode is a safety feature of the Macintosh power supply. The Macintosh is trying to start the machine and finding that the power requirements are greater than it can handle. The problem is that the power supply is not getting far enough into the startup procedure to turn itself off, so it keeps trying to turn itself on. The only way out of this mode is to pull the plug.

What’s Allowed and Why

Following are a few scenarios which might cause major heart problems for a user (these stories are fictional, and the names have been made generic to protect the innocent).

Slot	Card	Power Requirement
9	video card	10 watts
A	EtherTalk	10 watts
B	memory card	20 watts
C	A to D	15 watts
D	CPU	20 watts
E	video card	10 watts
Total		**85 watts**

This first scenario ends with a power requirement which exceeds the allowed power by 1.6 watts. The result of this over requirement can cause some very nasty results. Even if the machine could work, there is no guarantee to cover a thermal problem. The Macintosh was designed with the assumption that there would only be a need to dissipate 83.4 watts of NuBus power. If the machine must dissipate more than 83.4 watts of NuBus power, then it is possible that you might start burning chips.

An even worse scenario considers a fully loaded Macintosh IIcx. It is a lot easier to load up a IIcx, since the IIcx has half as many slots as the II and a power limit of 41.7 watts. This second scenario demonstrates a less high-powered user with a IIcx.

Slot	Card	Power Requirement
9	32-bit video card	15 watts
A	video card	10 watts
B	CPU	20 watts
Total		**45 watts**

In the second case, the machine is overdrawn by 3.3 watts. You may think that this is not a reasonable list of power requirements, but the reality of the power requirements is not the point. The point is that card developers must put forth an effort to protect the users, or we all look very bad when the silicon starts to melt. It is not very favorable to have our users burning up their machines because a NuBus card needed more power than it was allowed.

The wattage which a card requires is not the entire problem. It is possible to stay within the 13.9 watt limit and still have problems. You must also stay within the amperage limit for each voltage. You cannot just assume that since you are not using the 12 and -12 volts that you can use 2.78 amps of 5 volts (13.9 watts); the Macintosh power supply was not designed to convert 12 volt power allocation to 5 volt when it is needed. Scenario three presents an example of a Macintosh II which is filled with cards that are within the wattage limit, but that exceed the amperage limit.

Slot	Card	5 Volt Power Requirement	Amps
9	video card	10 watts	2
A	EtherTalk	10 watts	2
B	memory card	13.9 watts	2.78
C	A to D	13.9 watts	2.78
D	CPU	13.9 watts	2.78
E	video card	10 watts	2
Total		**71.7 watts**	**14.34**

Under normal conditions, the Macintosh II power supply can handle up to 12 amps at 5 volts. In the third scenario the NuBus cards are drawing 14.34 amps. Half of the cards are within the limit, but the other cards are not, and the result is a Macintosh which goes click-click.

But I Need the Power...

Now that we've told you not to take more power than you are allowed, we are going to give you a way out. We understand that it is impossible to fit within this power budget with some types of NuBus cards; if your card contains a processor, or worse, a lot of RAM, then you are going to run into the power allocation very quickly. In the rare case when you do need to consume the power of multiple slots, then you really must make absolutely sure that the slot or slots next to your card are not used.

The first possible solution is simply blocking off the slot or slots next to your board. You can build a device which extends out of your card to prevent the user from inserting other cards in the the adjoining slot or slots. The first slot to cover is the one on the component side of your card, thus allowing increased air flow on the side of your card which is most likely a little warm. This method, however, is not necessarily the method of choice. One of the problems with this method is that the power allocation is not part of the NuBus specification, it is a Macintosh-specific limit. It is always possible that this limit will be raised on future machines, and you do not want to implement this solution on machines where the problem is not a problem. The second solution is a bit cleaner than the first; however, it also has the potential for a similar problem with future machines.

The second solution is to design your card as a multiple-card implementation and have an internal bus which connects the two cards with ribbon cables or another type of connector. The benefits to this solution are a guarantee that users physically cannot put more cards in their systems than the power supply can handle and you get additional real estate with which to play.

A third, and perhaps simpler, solution is to ship a slot cover with your card. You can ask users to install the cover over the slot next to your card (or multiple slots if necessary). This cover should keep the user from inadvertently using the slot while not forcing the loss of a slot in any future machine with an increased power budget. This route would require an explanation and visible warning in the documentation; however, if the users do not heed your warning, then they cannot very well blame you for their clicking Macintosh (they will probably blame us).

These solutions are not the only ones which exist, but we haven't thought of any other great ideas. The main goal is to provide a method which protects users from overextending their machines. If you can devise such a method, then more power to you (well, not really).

Don't Get Flamed

So the moral (what's that) of the story is that you need to put yourself into the shoes of your users (but don't try it literally). If they burn up our computers or find themselves in click-click mode because a NuBus card got a little greedy, then they are going to be very upset, and that is something that both Apple and third-party developers need to work very hard to prevent. If you "need" the extra power, then you must make absolutely sure that users are not going to get burned by your NuBus card.

Further Reference:

- *Designing Cards and Drivers for the Macintosh*
- *IEEE Standard for a Simple 32-Bit Backplane Bus: NuBus*
- Technical Note #234, NuBus Physical Designs—Beware

NuBus is a trademark of Texas Instruments

Macintosh
Technical Notes

Developer Technical Support

#261: Cache As Cache Can

Revised by: Rich Kubota April 1992
Written by: Andrew Shebanow October 1989

This Technical Note documents cache behavior, manipulation of processor caches, and manipulation of external caches on Macintosh models that incorporate these features. It also describes how system software uses a memory management unit (when available) to implement special caching options.
Changes since October 1991: Described use of AppleTalk Transition Queue event, ATTransSpeedChange, when altering the 68040 cache state on the fly. This call must be issued so that LocalTalk can reevaluate its timers. Otherwise LocalTalk becomes disabled.

Cache Machines

The Motorola MC68020 microprocessor includes a 256-byte internal instruction cache. The MC68030 includes a similar-size instruction cache plus a 256-byte writethrough data cache. The MC68040 has much larger caches, 4K of instructions, and 4K of data. It also supports copyback caching in addition to the writethrough caching used by the MC68030.

The difference between writethrough and copyback caching is a matter of whether data writes go directly and immediately to main memory, or whether they go only as far as the data cache to be copied back to main memory later (if necessary) in a highly optimized fashion.

The MC68030 and MC68040 include memory management units internally. Besides the ability to divide memory into logical pages and provide memory access control, these memory management units can also associate cachability attributes with individual pages of memory, affecting how data is cached on a page-by-page basis.

Stale Data (Baked Fresh Daily)

Caching greatly improves overall system performance but introduces the problem of stale data, or inconsistency between cached data and the data in actual memory (RAM). In certain cases, cache maintenance instructions are necessary to maintain coherency between cache and main memory.

Stale Instructions

The first time when stale data becomes a problem occurs when writing self-modifying code on the MC68020 or any other processor with an instruction cache. The instruction cache remembers, separately from main memory, many of the instructions it has recently executed. If the processor executes an instruction, later changes that instruction in memory, and then tries to execute the new instruction at the same address, there is a probability that the original instruction is still cached. Since the cache is used before main memory, the old instruction may be executed instead of the new one, resulting in incorrect program operation.

To prevent this, any time a program changes an executable instruction in memory, it must flush the instruction cache before attempting to execute the modified instruction. Flushing a cache invalidates its entries, forcing the processor to refill cache entries from main memory. In part, this defeats the purpose of a cache and hurts performance. Nevertheless, the alternative is incorrect program operation. This serves to emphasize that caches must always be flushed judiciously in order to maintain correct operation and optimal performance.

As described, self-modifying code is not just code that changes itself directly as it executes. It can be much more subtle. Code that modifies a jump table entry is modifying executable code and must flush the instruction cache. Patch installation code often copies code from one block of memory into another and may modify one or more `JMP` instruction operands in order to get back to the original routine—either technique requires flushing the instruction cache.

Stale Data

With the addition of the data cache in the MC68030, performance is further enhanced, but another cache offers another source of stale data.

Let's say that you have a whizzy disk controller card that supports DMA. The board reads command buffers from the main CPU's memory area and writes status information back to the command buffer when done. Before the command is started, the MC68030 sets up the command buffer and zeroes the status code (the following figures are not to scale).

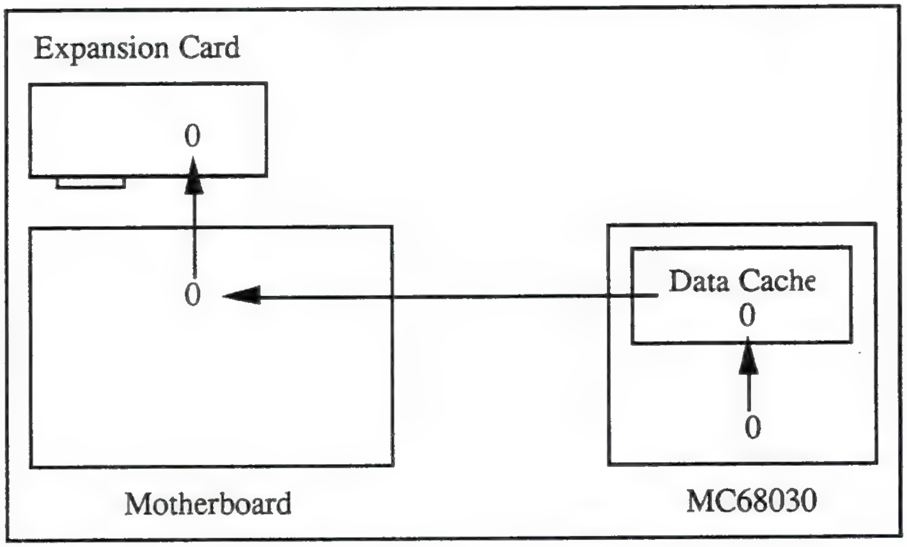


Figure 1 Write (Writethrough Cache)

At this point the cache and the memory both contain the value 0, since the MC68030's cache is writethrough (that is, it always writes data to memory immediately). Now the MC68030 starts the command running and waits for an interrupt from the disk controller card. It then reads back the status from the command buffer, which is modified by the DMA card.

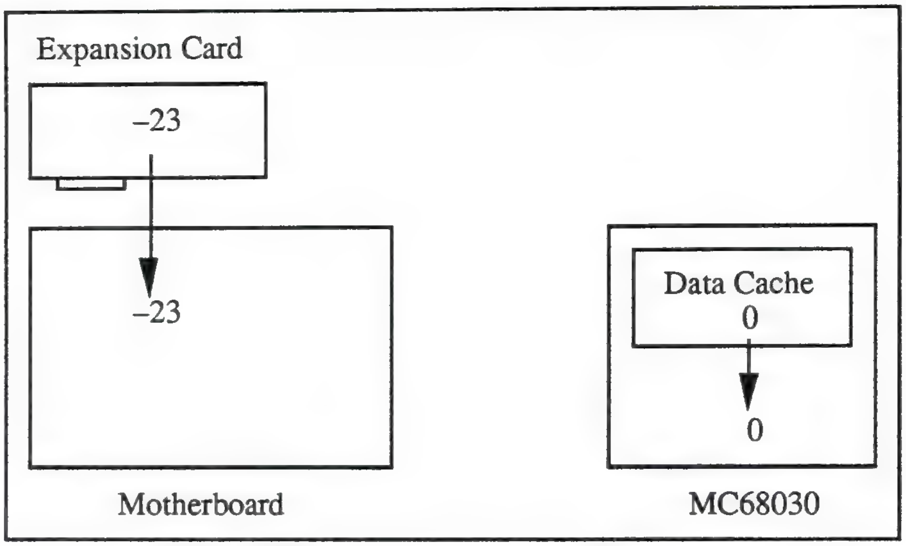


Figure 2 Read (From Cache)

Oops! Because the status code's value is already cached, the MC68030 thinks that the status is 0, even though the actual value in memory is –23. This type of thing can cause some very hard-to-find bugs in your driver.

Copyback Data and Stale Instructions

There is another type of cache called a copyback cache that is supported by more advanced microprocessors like the MC68040. A copyback cache further improves system performance by writing data to external memory only when necessary to make room in the cache for more recent accesses, or when explicitly "pushed" by system software. This is extremely valuable for relatively small, short-lived data that are accessed frequently but don't need to persist for a long time, like local stack frames in C and Pascal function calls.

This increase in performance again comes at some cost in terms of maintaining cache coherency. Here, the problem is twofold. Fundamentally, a datum that is "written to memory" isn't really in memory (meaning main RAM) until it's pushed out of the data cache. When performing DMA, it is necessary to push data cache contents into memory before instructing alternate bus masters to look for it; they'll only find stale data if it's still cached. Second, and perhaps even more important, the instruction and data caches are completely independent of each other. When fetching instructions, the processor looks in only two places: first the instruction cache, then main memory. It does not look in the data cache. When performing the types of operations described above that can cause a stale instruction cache, one must remember that it is impossible to make the instruction cache and memory coherent if memory itself is stale! The data cache must be flushed; then and only then can the instruction cache refill with the valid data the processor has written.

Here, some code writes the _LoadSeg trap to memory as part of a jump table update. Figure 3 indicates what happens if only the instruction cache is flushed. When execution later proceeds through that jump table entry, the processor fetches the opcode from that address and gets zonked with an illegal F-line exception. Why? _LoadSeg is still in the data cache. The code responsible for

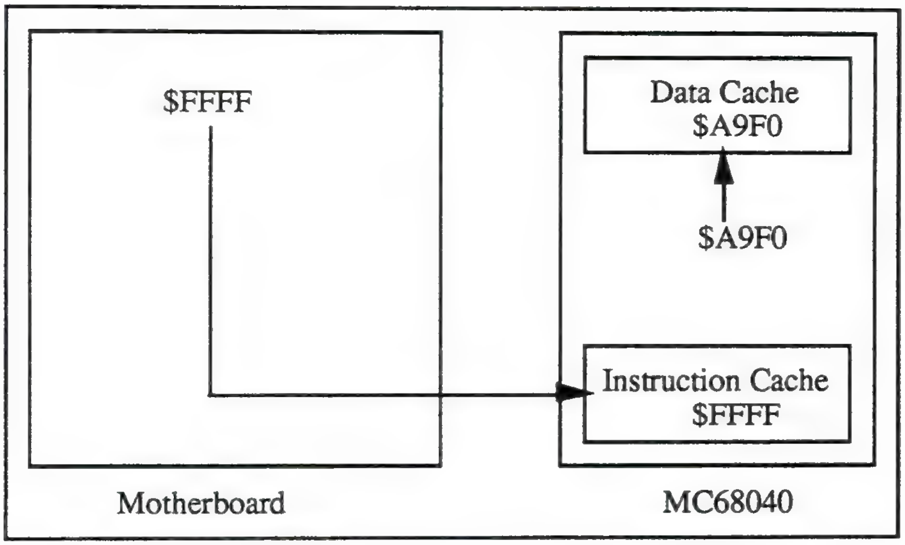


Figure 3 Write (Copyback Cache) and Fetch

maintaining the jump table failed to push the contents of the data cache before invalidating the instruction cache. This certainly causes problems on the MC68040.

Another similar problem applies to the time at which cache flushing is performed. When using a writethrough data cache, it is acceptable to invalidate the instruction cache first and then modify instructions in memory. With a copyback data cache, it is imperative to make changes to memory first and then flush caches. Again, this ensures that copyback data is written to memory before the instruction cache attempts to refill from memory. The key point to remember is that the MC68040 instruction cache always reads from memory, never from the data cache.

Figure 4 shows the path that an instruction properly takes when it is first written as data by a program that modifies instructions in memory.

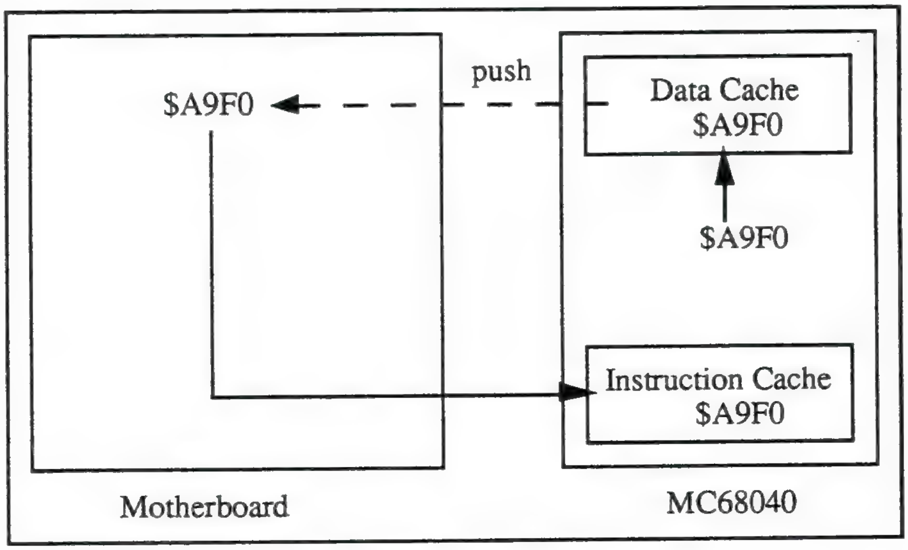


Figure 4 Write (Copyback Cache), Push, and Fetch

It's worth noting here that although pushing copyback data to memory and invalidating (flushing) the cache are conceptually different operations, they are at least for the MC68040 irrevocably connected. This makes flushing the data cache for the sake of pushing its contents to memory a potentially expensive one. Valid cache data is essentially lost when it is pushed and must be read from main memory if it is to be accessed again. This should be another reinforcement that cache flushing must be performed judiciously. It is possible to flush only a portion of the MC68040 caches, and software that flushes caches frequently should consider this optimization to avoid unnecessary performance degradation when running on this processor. See the interfaces provided below.

What Is Apple's Part in This?

There are two answers to this question. First, there are things that Apple has done in ROM to make life easier while dealing with a caching processor. Second, there are functions provided in ROM or in system software to allow developers to take some control of their own destinies.

Things That Happen for You

Ever since the Macintosh II made its debut, it has been flushing the instruction cache. It does so at a number of critical points where code may be moved to a new location, potentially leaving memory and the instruction cache incoherent. Specifically, there are a number of traps that have the potential to move code around memory. In each of these cases, the instruction cache is flushed by system software or ROM.

```
                    _BlockMove                    _LoadSeg
                    _Read                         _UnloadSeg
```

Warning: The _BlockMove trap is not guaranteed to flush caches for block sizes 12 bytes or less. This is a performance optimization. _BlockMove is called often by system software to move small blocks of memory that are not executable instructions. Flushing the cache in all such cases causes significant performance degradation. When moving very small blocks of code with _BlockMove, use one of the explicit cache flushing routines described below.

Note: C programmers should not assume that the standard library function memcpy() invokes _BlockMove. An explicit cache flush is required after moving code with memcpy().

In general, there may be others. As a rule of thumb, the instruction cache needs to be flushed explicitly only as a result of actions taken by user code, not as the result of anything a trap might have done. Traps can take care of themselves.

A memory management unit allows individual pages of memory to be marked noncachable. In current Macintosh implementations, NuBus™ and I/O address spaces are always marked noncachable—the processor won't cache memory stored at NuBus or I/O addresses. This solves any problems of stale data when processor/DMA "mailboxes" are located in NuBus memory and eliminates the fundamental problem of stale data at memory-mapped I/O locations. Data at RAM and ROM addresses are cachable, which makes sense and maximizes performance.

Since DMA still poses a problem when common buffers are located in main RAM, it would seem that there should be greater intrinsic support for specifying cachability. There is. In order for DMA masters to be compatible with abstract memory architectures like those defined by the Macintosh IIci and even more so by virtual memory, they must use the GetPhysical routine. Before using

GetPhysical, a range must always be locked with LockMemory. Since this sequence is so commonly required when performing DMA, the LockMemory routine has the effect of either disabling the data cache or marking the corresponding pages noncachable, depending on what's possible and what makes the most sense. In many cases, therefore, it is unnecessary to explicitly flush the data cache. If common DMA buffers are locked with LockMemory, the operating system ensures cache coherency at least for those buffers.

To ensure compatibility with existing code while taking advantage of copyback cache mode, the FlushInstructionCache function on an MC68040 actually flushes both caches using the CPUSHA BC instruction. This prevents the need for modification of correct existing code which properly flushes the instruction cache with FlushInstructionCache. If code is written properly for the MC68020 and MC68030, it will work on the MC68040 as well, without modification. If code is written incorrectly or directly manipulates the CACR register of these processors it will fail on the MC68040. When modifying code in memory or moving code about memory, use FlushInstructionCache before executing that code.

Facilities That Are Provided for You

Apple provides some system calls that let you flush the data and instruction caches without using privileged instructions (which is, as you should all know by now, a major no-no).

Following are the interfaces for these calls, for MPW Pascal and C (respectively):

```
FUNCTION SwapInstructionCache (cacheEnable: BOOLEAN) : BOOLEAN;
pascal Boolean SwapInstructionCache (Boolean cacheEnable);
```

This call enables or disables the instruction cache according to the state passed in cacheEnable and returns the previous state of the instruction cache as a result.

```
PROCEDURE FlushInstructionCache;
pascal void FlushInstructionCache (void);
```

This call flushes the current contents of the instruction cache. This has an adverse effect on CPU performance, so only call it when absolutely necessary.

```
FUNCTION SwapDataCache (cacheEnable: BOOLEAN) : BOOLEAN;
pascal Boolean SwapDataCache (Boolean cacheEnable);
```

This call enables or disables the data cache according to the state passed in cacheEnable and returns the previous state of the data cache as a result.

```
PROCEDURE FlushDataCache;
pascal void FlushDataCache (void);
```

This call flushes the current contents of the data cache. This has an adverse effect on CPU performance, so only call it when absolutely necessary.

Note: Before you call any of these routines, make sure that the _HwPriv ($A198) trap is implemented, or your program will crash. _HwPriv is implemented in the Macintosh IIx ROMs and later, as well as System 6.0.3 and later. The correct way to check for the trap is using the TrapAvailable function documented in *Inside Macintosh* Volume VI (pages 3-7 to 3-9).

These calls are provided as part of the MPW 3.1 library. For those of you without MPW 3.1 or later, you can use the following MPW assembly-language glue:

```
            CASE OFF

_HwPriv     OPWORD  $A198

SwapInstructionCache PROC EXPORT
            MOVEA.L (A7)+,A1        ; save return address
            MOVEQ   #0,D0           ; clear D0 before we shove Boolean into it
            MOVE.B  (A7)+,D0        ; D0 <- new mode
            MOVE.L  D0,A0           ; _HwPriv wants mode in A0
            CLR.W   D0              ; set low word to 0 (routine selector)
            _HwPriv
            MOVE.W  A0,D0           ; move old state of cache to D0
            TST.W   D0              ; if nonzero, cache was enabled
            BEQ.S   WasFalse        ; if zero, leave result false
            MOVEQ   #1,D0           ; set result to true
WasFalse:
            MOVE.B  D0,(A7)         ; save result on stack
            JMP     (A1)
            ENDPROC

FlushInstructionCache PROC EXPORT
            MOVEA.L (A7)+,A1        ; save return address
            MOVEQ   #1,D0           ; set low word to 1 (routine selector)
            _HwPriv
            JMP     (A1)
            ENDPROC

SwapDataCache PROC EXPORT
            MOVEA.L (A7)+,A1        ; save return address
            MOVEQ   #0,D0           ; clear D0 before we shove Boolean into it
            MOVE.B  (A7)+,D0        ; D0 <- new mode
            MOVE.L  D0,A0           ; _HwPriv wants mode in A0
            MOVE.W  #2,D0           ; set low word to 2 (routine selector)
            _HwPriv
            MOVE.W  A0,D0           ; move old state of cache to D0
            TST.W   D0              ; if nonzero, cache was enabled
            BEQ.S   WasFalse        ; if zero, leave result false
            MOVEQ   #1,D0           ; set result to true
WasFalse:
            MOVE.B  D0,(A7)         ; save result on stack
            JMP     (A1)
            ENDPROC

FlushDataCache PROC EXPORT
            MOVEA.L (A7)+,A1        ; save return address
            MOVEQ   #$3,D0          ; set low word to 3 (routine selector)
            _HwPriv
            JMP     (A1)
            ENDPROC
```

There are two additional calls whose interfaces follow. Each requires a little explanation.

The first call is `FlushCodeCache`, which simply invokes the `_CacheFlush` ($A0BD) trap. This trap's function is to make the instruction cache coherent with memory. On the MC68020 and MC68030 it simply flushes the instruction cache. On the MC68040 it also flushes the data cache for copyback compatibility. The advantage of `FlushCodeCache` as opposed to `FlushInstructionCache` is that it was implemented before the `_HwPriv` trap, and thus can be used on the Macintosh II while running older system software.

In general, `FlushInstructionCache` is still the preferred application-level cache flushing mechanism. `FlushInstructionCache` calls `FlushCodeCache` and is therefore a higher-level call conceptually. `FlushCodeCache` may be useful where `FlushInstructionCache` proves unsuitable, or as an alternative to the next call, FlushCodeCacheRange. Obviously, before calling FlushCodeCache, be certain that _CacheFlush is implemented.

Note: If the processor has a cache to flush, this trap should be properly implemented, because ROM and system software use this trap's vector to do their own cache flushing. In fact, FlushInstructionCache itself uses this vector. This should be of particular interest to accelerator card developers.

```
                  MACRO
                _FlushCodeCache
                _CacheFlush
                ENDM

PROCEDURE FlushCodeCache;
    INLINE $A0BD;

void FlushCodeCache (void) = 0xA0BD;
```

The second call is FlushCodeCacheRange. FlushCodeCacheRange is an optimization of FlushCodeCache designed for processors like the MC68040 which support flushing only a portion of the cache. (The MC68020 and MC68030 do not support this feature and FlushCodeCacheRange simply flushes the entire instruction cache on those processors.) As described earlier, pushing and flushing cache entries are linked and flushing the entire cache after a small change like a jump table entry can be expensive. FlushCodeCacheRange allows one to request that only a specific memory range be flushed, leaving the rest of the cache intact. Note that this is only a request and that more than the requested range may be flushed if it proves inefficient to satisfy the request exactly. Also, FlushCodeCacheRange may not be implemented for some older versions of system software that are not MC68040-aware. If not, FlushCodeCacheRange returns hwParamErr (–502) and it is necessary to flush the entire cache instead, probably using FlushCodeCache. If FlushCodeCacheRange succeeds it returns noErr (0). Before calling FlushCodeCacheRange, be certain that _HwPriv is implemented.

```
; _FlushCodeCacheRange takes/returns the following parameters:
;     -> A0.L =   Base of range to flush
;     -> A1.L =   Length of range to flush
;     <- D0.W =   Result code (noErr = 0, hwParamErr = -502)
                MACRO
                _FlushCodeCacheRange
                moveq   #9,d0
                _HwPriv
                ENDM

FUNCTION FlushCodeCacheRange (address: UNIV Ptr; count: LongInt) : OSErr;
    INLINE $225F,   { MOVEA.L (SP)+,A1 }
           $205F,   { MOVEA.L (SP)+,A0 }
           $7009,   { MOVEQ   #9,D0    }
           $A198,   { _HwPriv          }
           $3E80;   { MOVE.W  D0,(SP)  }

// MPW C 3.2 makes register-based inline calls very efficient.
#pragma parameter __D0 FlushCodeCacheRange(__A0,__A1)
OSErr FlushCodeCacheRange (void *address, unsigned long count) =
    {0x7009, 0xA198};

/* MPW C 3.1 and earlier, and THINK C™ should declare the function as   */
/* "pascal" and use the same inline constants as the Pascal interface: */
pascal OSErr FlushCodeCacheRange (void *address, unsigned long count) =
    {0x225F, 0x205F, 0x7009, 0xA198, 0x3E80};
```

Caching Consequences—LocalTalk

As noted above, altering the state of the data/code cache significantly affects the performance of the 68040 processor. This change in effective CPU speed may affect any background process that is dependent on the processor speed remaining constant. LocalTalk is an example of one such affected process, as it employs speed sensitive timing loops. The change in CPU speed affects the LocalTalk timers, to the extent that the LocalTalk no longer functions correctly if it is the current AppleTalk connection.

Fortunately, the AppleTalk Transition Queue mechanism can be used to notify LocalTalk of the change in effective CPU speed. Upon notification, LocalTalk recalculates its timer values to match the current CPU speed. Refer to *Inside Macintosh* Volume VI, page 32-17, and to Technical Note #311, "What's New In AppleTalk Phase 2," for additional information on the use of the AppleTalk Transition Queue.

The following code demonstrates the use of the `ATEvent` procedure to send the `ATTransSpeedChange` event. The `ATEvent` call is provided as part of the MPW 3.2 library. **Important Note**: Issue the `ATTransSpeedChange` event **only** at `SystemTask` time!

```
USES AppleTalk;                            { ATEvent prototyped in AppleTalk unit, MPW 3.2 }

CONST
    ATTransSpeedChange = 'sped';           {change in cpu speed transition }

PROCEDURE NotifyLocalTalkSpeedChange;

    BEGIN
        if LAPMgrExists THEN               { check LAP Manager exists, see Tech Note 311 }
                                           { for the code for LAPMgrExists }
            ATEvent(longint(ATTransSpeedChange), NIL);
                                           { notify speed change event }
    END;
```

Note that only LocalTalk drivers that are included with AppleTalk version 57 or greater, respond to the ATTranSpeedChange event. System 7.0.1 for the Quadra's, is supplied with AppleTalk version 56. AppleTalk version 57 is available by using the AppleTalk Remote Access Installation program, or the Network Software Installer version 1.1. Licensing for AppleTalk can be arranged by contacting Apple Software Licensing. Software Licensing can be reached as follows:

Software Licensing
Apple Computer, Inc.
20525 Mariani Avenue, M/S 38-I
Cupertino, CA 95014
MCI: 312-5360
AppleLink: SW.LICENSE
Internet: SW.LICENSE@AppleLink.Apple.com
(408) 974-4667

AppleTalk version 53 or greater is required to handle the ATEvent call, however, nothing bad will happen if you issue the ATTranSpeedChange transition event under AppleTalk versions 53 - 56. It is important to check that the LAP Manager is implemented before issuing the `ATEvent` call. See Tech Note 311 for a description of the `LAPMgrExists` function.

External Caches

The Macintosh IIci and Macintosh IIsi support external cache cards. Because of the way these caches work, cache coherency is not much of a problem. In fact these caches are usually enabled full-time and their operations are totally transparent to all well-behaved hardware and software. Still, there are corresponding cache control functions to enable, disable, and flush these cache cards. If _HwPriv is implemented, the following routines may be used:

```
                 MACRO
               _EnableExtCache
               moveq    #4,d0
               _HwPriv
               ENDM

PROCEDURE EnableExtCache;
    INLINE $7004,$A198;

void EnableExtCache (void) = {0x7004, 0xA198};


               MACRO
               _DisableExtCache
               moveq    #5,d0
               _HwPriv
               ENDM

PROCEDURE DisableExtCache;
    INLINE $7005,$A198;

void DisableExtCache (void) = {0x7005, 0xA198};


               MACRO
               _FlushExtCache
               moveq    #6,d0
               _HwPriv
               ENDM

PROCEDURE FlushExtCache;
    INLINE $7006,$A198;

void FlushExtCache (void) = {0x7006, 0xA198};
```

Further Reference:

- *Inside Macintosh*, Volume V, Operating System Utilities
- *Inside Macintosh*, Volume VI, Compatibility Guidelines
- *Designing Cards and Drivers for the Macintosh Family*
- *M68000 Family Programmer's Reference Manual*
- *M68020 32-Bit Microprocessor User's Manual*
- *M68030 Enhanced 32-Bit Microprocessor User's Manual*
- *M68040 32-Bit Third-Generation Microprocessor User's Manual*

NuBus™ is a trademark of Texas Instruments.
THINK C is a trademark of Symantec Corp.

Macintosh
Technical Notes

Developer Technical Support

#262: Controlling Status Calls

Written by: Tim Enwall February 1990

This Technical Note discusses situations under which high-level `Status` calls do not work correctly and `PBStatus` calls should be made instead.

When Apple designed the `_Status` trap, it was assumed that device drivers would only **return** status information and not require any information other than the `csCode` to determine what information to return. The device driver was supposed to return the information in the variable `csParam`. However, some recently designed drivers **do** depend on information being sent to the device driver's status routine, and hence `csParam` must be valid.

The glue code in many development environments builds the parameter block which gets passed to `_Status` calls. The high-level `Status` call passes only the `refNum`, `csCode` and `csParamPtr`, but the glue neglects to fill in the `csParam` field of the parameter block because of the above assumption. The low-level `PBStatus` call has no such problem because `PBStatus` passes the entire parameter block which **you** must fill in, and hence you assign the `csParam` field correctly. Thus the high-level `Status` call in some cases either works incorrectly, or worse, causes problems for the device driver.

The most obvious example of a device driver which expects a valid `csParam` is the video device driver(s) for Macintosh II Video Cards. Almost all of the documented status calls require `csParam` to point to some kind of table. In this case, most of the device driver's status routines do not function properly **if** using the high-level `Status` call.

Therefore, if you are interfacing to a device driver which you either know or suspect requires `csParam` for its status calls, use the low-level `PBStatus` call instead of the high-level `Status` call. If you are writing a device driver, be aware of this limitation and either alert the users of your driver to this limitation, or design your status calls to only **return** `csParam`.

Further Reference:

- *Inside Macintosh*, Volume II, The Device Manager

#263: International Canceling

Written by: John Harvey

February 1990

This Technical Note describes potential problems canceling operations with the Command-period key sequence and international keyboards.

Where Did That Key Go?

Canceling an operation, from printing to compiling, has always been done with the key sequence Command-period. The problem with this is that on some international systems, one needs to hold the Shift key down to produce a period. Many keyboard mappings, including that of the U.S., ignore the Shift key when the Command key is down. In other words, on a system where a period (.) is a shifted character (e.g., Italian) pressing Command-Shift-KeyThatMakesAPeriod does not generate the ASCII code for a period. Instead, the keyboard mapping software generates the ASCII code for the unshifted character. If an application is looking for Command-period to cancel some time intensive operation, and an international user types the shifted key sequence that normally produces a period along with the Command key, the application is going to miss that request unless it takes special precautions.

A Bit Confusing (to me at least)

The solution to this potential international disaster is to strip the Command key out of the modifiers, and then run the key code back through the keyboard mapping software. The trap `_KeyTrans` makes this procedure very easy. `_KeyTrans` takes as parameters a pointer to a `'KCHR'` resource (see Technical Note #160, Keyboard Mapping), a word which contains the keycode and the modifier bits, and a word which serves as a state variable.

One note on the result returned by `_KeyTrans`. *Inside Macintosh*, Volume V-195, The Toolbox Event Manager, states, "ASCII 1 is the ASCII value of the first character generated by the key code parameter." This statement is followed by an illustration (Figure 7 on page V-195) which shows ASCII 1 as the low byte of the high word in the long word result. Although this statement and the accompanying illustration are correct, they have mislead a number of people (me for one).

It is dangerous to expect the character code in one particular word of the long word result. In fact, the architecture of the `_KeyTrans` trap does not specify which word contains the character code in which you might be interested. This is because the `_KeyTrans` trap's primary purpose is to create a package that can be used to build a key-down event, and the Toolbox Event Manager just doesn't care about particular keys. In fact, it is possible to get a result from `_KeyTrans` that contains character codes in both words. This is how dead keys are handled.

But how does one handle a particular character, specifically a period? The strategy adopted in the sample function in this Note is to check both words of the result. If a period exists in either word and the Command key is down, it is counted as a Command-period key sequence.

Now that everything is straight about parameters and results, it's time to look at some sample code. The code fragment which follows ensures that you get that period regardless of the state of the modifier keys.

MPW Pascal

```
CONST
  kMaskModifier = $FE00; {need to strip command key from Modifiers}
  kMaskVirtualKey = $0000FF00; {get virtual key from event message}
  kMaskASCII1 = $00FF0000;
  kMaskASCII2= $000000FF; {get key from KeyTrans return}
  kKeyUpMask  = $0080;
  kPeriod = ORD('.');

TYPE
  EventPtr = ^EventRecord;

FUNCTION CmdPeriod(theEvent: EventPtr): Boolean;
VAR
  keyCode     : Integer;
  virtualKey,
  keyInfo,
  lowChar,
  highChar,
  state,
  keyCId      : Longint;
  hKCHR       : Handle;

BEGIN

  CmdPeriod  := FALSE;

  IF ( theEvent^.what = keyDown ) | ( theEvent^.what = autoKey ) THEN BEGIN

    {see if the command key is down.  If it is, get the ASCII }
    IF  BAND(theEvent^.modifiers,cmdKey) <> 0  THEN BEGIN

      virtualKey := BAND(theEvent^.message,kMaskVirtualKey) DIV 256;
    {strip the virtual key by ANDing the modifiers with our mask}
      keyCode := BAND(theEvent^.modifiers,kMaskModifier);
      keyCode := BOR(keyCode,kKeyUpMask);  {let KeyTrans think it was a keyup event, this
                                            will keep special dead key processing from
                                            occurring }
    {Finally OR in the virtualKey}
      keyCode := BOR(keyCode,virtualKey);
      state := 0;

      keyCId := GetScript( GetEnvirons(smKeyScript), smScriptKeys);

      {read the appropriate KCHR resource }
      hKCHR := GetResource('KCHR',keyCId);

      IF hKCHR <> NIL THEN BEGIN
        { we don't need to lock the resource since KeyTrans will not move memory }
        keyInfo := KeyTrans(hKCHR^,keyCode,state);
        ReleaseResource(hKCHR);
      END
      ELSE
        {if we can't get the KCHR for some reason we set keyInfo to the message field.  This
         ensures that we still get the Cancel operation on systems where '.' isn't shifted.}
        keyInfo := theEvent^.message;

      LowChar := BAND(keyInfo,kMaskASCII2);
      HighChar := BSR(BAND(keyInfo,kMaskASCII1),16);

      IF ( LowChar = kPeriod ) | (HighChar = kPeriod) THEN
           CmdPeriod := TRUE;
    END;
  END;
END;
```

MPW C

```
#define kMaskModifiers  0xFE00 // we need the modifiers without the command key for KeyTrans
#define kMaskVirtualKey 0x0000FF00 //get virtual key from event message for KeyTrans
#define kUpKeyMask      0x0080
#define kShiftWord      8 //we shift the virtual key to mask it into the keyCode for KeyTrans
#define kMaskASCII1     0x00FF0000 // get the key out of the ASCII1 byte
#define kMaskASCII2     0x000000FF  //get the key out of the ASCII2 byte
#define kPeriod         0x2E // ascii for a period

Boolean CmdPeriod( EventRecord *theEvent )
{

  Boolean  fTimeToQuit;
  short    keyCode;
  long     virtualKey, keyInfo, lowChar, highChar, state, keyCId;
  Handle   hKCHR;

  fTimeToQuit = false;

  if (((*theEvent).what == keyDown) || ((*theEvent).what == autoKey)) {

  // see if the command key is down.  If it is, find out the ASCII
  // equivalent for the accompanying key.

  if ((*theEvent).modifiers & cmdKey ) {

    virtualKey = ((*theEvent).message & kMaskVirtualKey) >> kShiftWord;
    // And out the command key and Or in the virtualKey
    keyCode    = ((*theEvent).modifiers & kMaskModifiers)  |  virtualKey;
    state      = 0;

    keyCId     = GetScript( GetEnvirons(smKeyScript), smScriptKeys );
    hKCHR      = GetResource( 'KCHR', keyCId );

    if (hKCHR != nil) {
      /* Don't bother locking since KeyTrans will never move memory */
      keyInfo = KeyTrans(*hKCHR, keyCode, &state);
      ReleaseResource( hKCHR );
    }
    else
     keyInfo = (*theEvent).message;

    lowChar =  keyInfo &  kMaskASCII2;
    highChar = (keyInfo & kMaskASCII1) >> 16;
    if (lowChar == kPeriod || highChar == kPeriod)
      fTimeToQuit = true;

  }  // end the command key is down
}  // end key down event

return( fTimeToQuit );
}
```

What About That Resource

The astute observer may have noticed that the code example requires that you read a resource. Although this certainly isn't that big of a deal, it is always nice when you can cut down on disk accesses. In System 7.0 a verb is added that can be used to get _GetEnvirons to return a pointer to the current 'KCHR'. The verb is defined and used as follows:

Pascal

```
CONST
    smKCHRCache =  38;

    KCHRPtr := GetEnvirons(smKCHRCache);
```

C

```
#define smKCHRCache 38

    KCHRPtr = GetEnvirons(smKCHRCache);
```

Unfortunately, in system software prior to 7.0, you must use _GetResource as demonstrated above to obtain the current 'KCHR' resource. However, since _GetEnvirons always returns zero when passed a verb it does not recognize, you can build System 7.0 compatibility into your application without having to check which system software is running. To do this, you could modify the routines as follows:

Pascal

```
CONST {define our own constant until System 7.0 headers ship.  At that point, if you
       have not shipped, you can put in the real constant}
    NewVerb_smKeyCache = 38;
VAR
    KCHRPtr : Ptr;

    KCHRPtr := Ptr(GetEnvirons(NewVerb_smKeyCache ));
    hKCHR   := NIL;  {set to NIL before starting}

    IF KCHRPtr = NIL THEN BEGIN  {we didn't get the ptr from GetEnvirons}
      keyCId := GetScript(GetEnvirons(smKeyScript), smScriptKeys);

      {read the appropriate KCHR resource }
      hKCHR := GetResource('KCHR',keyCId);
      KCHRPtr := hKCHR^;
    END;

    IF KCHRPtr <> NIL THEN BEGIN
      { we don't need to lock the resource since KeyTrans will not move memory }
      keyInfo := KeyTrans(KCHRPtr,keyCode,state);
      IF hKCHR <> NIL THEN
        ReleaseResource(hKCHR);
    END
```

C

```
/* again we define our own constant for now */
#define NewVerb_smKeyCache 38

Ptr KCHRPtr;

    hKCHR = nil;  /* set this to nil before starting */
    KCHRPtr = (Ptr)GetEnvirons(NewVerb_smKeyCache );

    IF ( !KCHRPtr ) {
      keyCId = GetScript( GetEnvirons(smKeyScript), smScriptKeys);

      hKCHR   = GetResource('KCHR',keyCId);
      KCHRPtr = *hKCHR;
    };

    IF (KCHRPtr) {
      keyInfo := KeyTrans(KCHRPtr ,keyCode,state);
      if (hKCHR)
        ReleaseResource(hKCHR);
    }
```

Further Reference:

- *Inside Macintosh*, Volume V, The Script Manager
- *Inside Macintosh*, Volume V, The Toolbox Event Manager
- Technical Note #160, Key Mapping

Macintosh
Technical Notes

Developer Technical Support

#264: Script Manager 2.0 Date & Time Problems

Written by: John Harvey February 1990

This Technical Note describes known bugs and features in and solutions to the date and time routines introduced in Script Manager 2.0.

From the beginning, the Macintosh's ability to handle dates was limited to a rather small range—slightly more than a century. Enhancements to the Script Manager, introduced with System Software 6.0, extended this range to ±35,000 years. Unfortunately, there is a minor bug in one of the crucial calls and a "feature" that looks like a bug in another.

You Said It Would Be A Long Time

_LongSecs2Date, the routine that translates a LongDateTime to a LongDateRec, has a bug caused by using a variable that has not been properly initialized. This bug rears its ugly head when negative values are passed to the routine. System Software 6.0.4 and later fix this bug, and there is a simple solution for earlier systems.

If using System Software 6.0.3 and earlier, if you call _LongSecs2Date once before you really want to use it, the variable is cleared. After the initial call, _LongSecs2Date works correctly.

For example:

MPW Pascal

```
PROCEDURE DoDateStuff;
VAR
  lsecs: LongDateTime;
  ldr:   LongDateRec;
BEGIN
  InitDateCache(dcr);
  lsecs := 0;
  LongSecs2Date(lsecs,ldr);
  {now you can call LongSecs2Date for real}

END;
```

MPW C

```
void DoDateStuff()
{
  LongDateTime lsecs;
  LongDateRec  ldr;


/* work around the bug */
  lsecs = 0;
  LongSecs2Date(&lsecs,&ldr);
/* now call LongSecs2Date for real */
}
```

Any String To Date

The routine _String2Date was originally designed to be as forgiving as possible. It is so forgiving that it accepts any non-alphabetic character as a separator and accepts a single number as a valid date. For instance, if you pass _String2Date a string like "<20" it generously assumes that the less than sign (<) is intended as a divider and that "20" must be intended as a day, since there are only 12 months in a year. It returns a result of noErr and a date which is the twentieth of the current month in the current year. The string "<3*3" produces March 3 of the current year, while "4>1" politely gives the date April 1 of the current year.

This forgiveness really is not a bug, but a feature. Unfortunately it isn't a feature that has been greatly appreciated in the developer community. For that reason, the rules for date and time dividers are tighter in System 7.0. Current thinking is that all list separators now used in 'itl0' resources will be allowed with a few common date separators used in the U.S. (e.g., colon (:) and hyphen (-)). For now, it is important to be aware of, shall we say, the flexibility of _String2Date and avoid thinking of it as an intelligent date parser. If you want to parse something, you can use _IntlTokenize.

Further Reference:

- *Inside Macintosh*, Volume V, The Script Manager
- *The Script Manager 2.0*, Interim Chapter (DTS)

Macintosh
Technical Notes

Developer Technical Support

#265: Pascal to C: PROCEDURE Parameters

Revised by: Keith Rollin — August 1990
Written by: Keith Rollin — February 1990

This Technical Note talks about nested procedures and PROCEDURE parameters in Pascal and what to do when converting them into C or C++.
Changes since February 1990: Fixed some type coercion problems.

Pascal and C offer many of the same features, but there are some differences. These differences make converting between languages or calling libraries written in one language from the other difficult sometimes. Two closely associated features of Pascal that C does not offer are nested procedures and PROCEDURE parameters. Since these two features are commonly used when programming with MacApp, the problem of implementing them in C++ is a common one.

How Pascal Implements Nested Procedures

Pascal lets programmers nest procedures within each other. Doing so allows one to limit the scope of local variables, as well as allow multiple procedures access to the same set of dynamically created variables.

Let's take a look at the following bit o' code:

```
PROCEDURE CallBack;                  { Outer level procedure }

  BEGIN
  END;

PROCEDURE CallingProcedure;          { Outer level procedure }

  VAR
    aVar: integer;

  PROCEDURE NestedCallBack;          { Nested procedure - can access "aVar" }

    VAR
      anotherVar: integer;

    BEGIN {NestedCallBack }
      aVar := 1;
      anotherVar := 2;
    END; {NestedCallBack }

  BEGIN {CallingProcedure }
    CallBack;
    NestedCallBack;
  END; {CallingProcedure }
```

This code shows three Pascal procedures: CallingProcedure, CallBack, and NestedCallBack. NestedCallBack is the nested procedure, which means that it can access the local variables of the procedure it is nested within, namely, CallingProcedure.

The method used to allow NestedCallBack to access its host procedure's local variables is not so obvious and involves a little hack. As you may know, local variables are created on the stack when a procedure is entered, and the 680x0 register A6 is initialized to point to them. Fine, but this leads to a little conflict within NestedCallBack. It needs to use A6 to point to its own local variables (e.g., anotherVar), so how does it access its host procedure's local variables?

The answer, logically enough, is that it uses another register for this purpose. When NestedCallBack is called from its host procedure, the host's A6 is pushed onto the stack after any and all formal parameters have been pushed on, but before the JSR is performed (this extra parameter is often referred to as the "static link"). As NestedCallBack is being entered, you have a stack similar to that in Figure 1. By comparison, Figure 2 shows what the stack would look like if you made a normal call to a procedure on the outer level, such as the procedure shown above named CallBack.

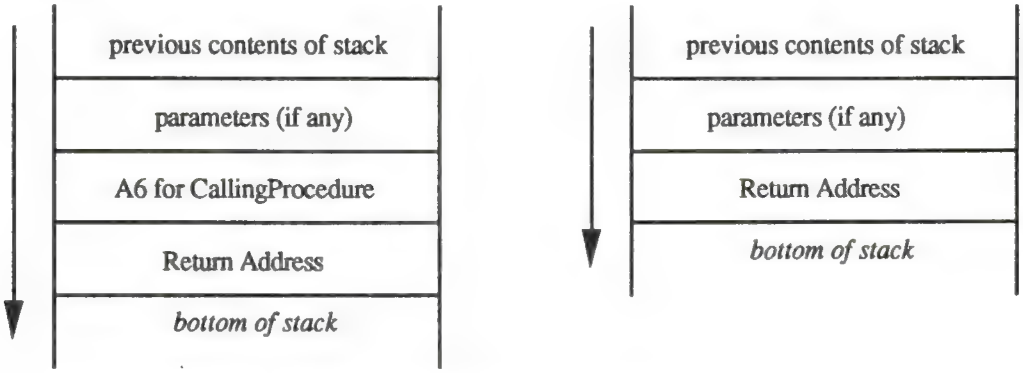


Figure 1–Call to NestedCallBack & Figure 2–Call to CallBack

Each procedure knows at compile time whether it is nested or not and adjusts itself accordingly. If it turns out that a procedure is nested, then it is compiled as if you had declared an extra parameter at the end of the formal parameter list, one that held the value of the host's A6. Pascal then uses this parameter for fetching the local variables of the nested procedure's host. It pulls this parameter off of the stack just like any other parameter, sticks it into a handy register, and uses it as a base address to the host's local variables, just as it uses A6 as the base address to its own locals.

How Pascal Implements PROCEDURE Parameters

As seen in the previous section, nested procedures require a little help to get themselves up and running. Specifically, they need an extra parameter called a static link. You've seen one way in which Pascal provides support for this parameter. In this section, you see another important case.

There are many Toolbox routines that require a pointer to a procedure being passed to them as a parameter. These procedures are called "callback" procedures, because the Toolbox makes a call back to those procedures to perform some application-specific function. An example of this type of routine would be the Control Manager routine _TrackControl, which requires a callback procedure called actionProc.

By now, you should see why you cannot pass the address of a nested procedure to such a Toolbox routine. Nested procedures require that they be passed the static link parameter so that they can access their host variables. The Toolbox doesn't support this convention, so it cannot pass the required static link to the nested routine.

While the Toolbox doesn't support the nested procedure convention, Pascal itself does support a method whereby you can pass around all the information necessary to implement a callback procedure as a nested procedure. Syntactically, this is done by including a full procedure heading in the list of formal parameters a procedure takes. An example of such could look like the following:

```
PROCEDURE SomeProcedure(PROCEDURE CallBackProc(i:integer); iterForward: BOOLEAN);
  BEGIN
    ...
    CallBackProc(5);
    ...
  END;
```

`SomeProcedure` takes two parameters. The first is a `PROCEDURE` parameter that refers to a routine that takes a parameter itself, namely, a single integer. In addition, `SomeProcedure` takes a `BOOLEAN` called `iterForward` as a second parameter. You would call `SomeProcedure` with something like the following:

```
PROCEDURE MyCallingProcedure;

  PROCEDURE MyCallBackProcedure(i: integer);

    BEGIN { of MyCallBackProcedure }
      < mumble >;
    END;

  BEGIN { of MyCallingProcedure }
    SomeProcedure(MyCallBackProcedure, TRUE);
  END;
```

Through the use of the `PROCEDURE` parameter, you can invoke the callback procedure using a natural Pascal syntax. In the `SomeProcedure` example, the statement `CallBackProc(5)` causes `MyCallBackProcedure` to be called with a value of five. Not only can you just invoke the procedure by entering the name of the `PROCEDURE` variable, but you can pass parameters to it with full Pascal typechecking invoked. In this case, Pascal ensures that when you call `CallBackProc`, you also pass a single integer to it.

`PROCEDURE` parameters also give the support for nested procedures for which you are looking. When a `PROCEDURE` parameter is passed on the stack, two components are used to represent it. The first is a pointer to the actual procedure. The second is the static link. Therefore, you can think of a `PROCEDURE` parameter as being represented by the following record:

```
TYPE
  ProcedureParameter = RECORD
    procPtr: Ptr;
    staticLink: Ptr;
  END;
```

When you pass a `PROCEDURE` parameter to a destination procedure, both of these components are pushed onto the stack as `LONG` values (four bytes each). When it comes time for the destination procedure to invoke the callback, any necessary parameters for the callback are placed onto the stack, followed by the `staticLink` value. Then the routine specified by `procPtr` is called.

The step where the destination procedure pushes the static link onto the stack is important and should be examined more closely. Specifically, how do you know that a static link parameter is necessary at this point? After all, SomeProcedure simply declares that it takes a PROCEDURE as a parameter; it doesn't differentiate between nested and non-nested procedures. But, as you saw in the first section, these two kinds of procedures are called differently. How do you know if the the static link passed to you needs to be pushed onto the stack for the callback procedure?

The answer is that SomeProcedure receives a special value for the static link parameter for non-nested procedures. If the callback procedure is at the outer level, SomeProcedure receives NIL for the value of the static link. When Pascal compiles the commands that invoke PROCEDURE parameters, it generates code that checks the static link. If it is NIL, it doesn't push it onto the stack. If it is not NIL, then you are calling a nested procedure, and must push the static link onto the stack.

So, how do you utilize nested procedures and PROCEDURE parameters in C or C++? Obviously, you cannot—at least not directly. C and C++ don't support them. At this point, you might as well just give up and use Pascal; you always said C++ was highly overrated anyway.

There are two scenarios to examine:

- A Pascal routine calls your C++ routine, passing a PROCEDURE parameter to another Pascal routine you have to call.
- Your C++ routine calls a Pascal routine expecting a PROCEDURE parameter, which you have implemented in C++.

The rest of the Note looks at both of these cases.

Pascal to C++ to Pascal

MacApp supports an object inspector, which it implements by calling a Fields method common to all descendants of TObject. Each class you define should override this method so that MacApp can find out about your class's fields. Such a method definition would look like the following:

```
PROCEDURE TJustCommand.Fields(PROCEDURE DoToField(fieldName: Str255; fieldAddr: Ptr;
                                                  fieldType: INTEGER)); OVERRIDE;

BEGIN
  DoToField('TJustCommand', NIL, bClass);
  DoToField('fTEView', @fTEView, bObject);
  DoToField('fOldJust', @fOldJust, bInteger);
  DoToField('fNewJust', @fNewJust, bInteger);
  INHERITED Fields(DoToField);
END;
```

You tell it the name of your class so that whatever routine is calling you (usually MacApp's inspector or debugger) can identify the class it is inspecting. Then, for each field in your class, you call the procedure passed to you, giving it the three parameters it needs. Finally, you call your superclass' Fields method so that it can identify its name and fields.

When your `Fields` method is called, the `DoToField` parameter appears on the stack as a pointer to the procedure you are supposed to call, as well as the static link value it needs. When you actually call `DoToField`, the necessary parameters are first pushed onto the stack (i.e., `fieldName`, `fieldAddr`, and `fieldType`). Pascal then adds some code that makes a determination based on the value of the static link parameter. If it is non-zero, then you are calling a nested procedure and need to pass back the static link back on the stack. If static link is zero, then you are not calling a nested procedure and don't need to pass that static link back.

Pascal handles all of this for you transparently. This ease in Pascal makes the process of writing a similar routine in C or C++ that much more difficult, as that process has been hidden from us.

There is no way in C or C++ to pass a variable number of parameters in one statement. In other words, you **cannot** do something like the following:

```
DoToField("\pTJustCommand", nil, bClass, StaticLink ? StaticLink : void); /* No Workie */
```

That would be too easy. Instead, you must use some inline glue that prepares the stack for you. This inline procedure accepts the three parameters you see in the Pascal version, as well as both components of the `PROCEDURE` parameter (i.e., the procedure pointer and static link). The glue looks at the static link value and removes it from the stack if it is zero and, thus, not needed.

One solution is as follows:

```
typedef pascal void (*FieldProcPtr) (StringPtr fieldName, Ptr fieldAddr,
                                     short fieldType, void *DoToField_StaticLink);

pascal void CallDoToField(StringPtr, Ptr, short, void *, FieldProcPtr)
  = {
    0x205F,     // MOVEA.L    (A7)+,A0     ; get the DoToField pointer
    0x4A97,     // TST.L      (A7)         ; check the StaticLink
    0x6602,     // BNE.S      *+$0004      ; if non-zero, keep it in
    0x588F,     // ADDQ.L     #$4,A7       ; if zero, pull it off
    0x4E90      // JSR        (A0)         ; Call DoToField
    };

pascal void TJustCommand::Fields(FieldProcPtr DoToField, void *DoToField_StaticLink)
{
  CallDoToField("\pTJustCommand", NULL, bClass, DoToField_StaticLink, DoToField);
  CallDoToField("\pfTEView", (Ptr) &fTEView, bObject, DoToField_StaticLink, DoToField);
  CallDoToField("\pfOldJust", (Ptr) &fOldJust, bInteger, DoToField_StaticLink, DoToField);
  CallDoToField("\pfNewJust", (Ptr) &fNewJust, bInteger, DoToField_StaticLink, DoToField);
  inherited::Fields(DoToField, DoToField_StaticLink);
```

C++ to Pascal to C++

Now look at another case that occurs often in MacApp. This is where your C++ routine calls a MacApp procedure that needs a `PROCEDURE` reference back to one of your own routines. For instance, MacApp has a class called `TList` that allows you to maintain a list of objects. This class has a method called `Each` that allows you to perform some operation on each object in the list. MacApp takes care of iterating over all of the objects and calls a routine you pass to it for each one.

For this example, you have a list of objects stored in a `TList` and you want to pass the `Graze` message to all of them. At the same time, you want to keep track of how many grazed so much that they fell off a cliff during the process. If the number of objects grazing off a cliff is greater than some threshold, then you call `_SysBeep`. You could use the following procedures to accomplish this in Object Pascal:

```
  VAR
    myList: TList;

  PROCEDURE TMyApplication.GrazeAll;

    VAR
      offTheCliff: integer;

    PROCEDURE DoGraze(theObject: TObject);

      BEGIN
        TGrazer(theObject).Graze;
        IF TGrazer(theObject).GrazedOffTheCliff THEN
          offTheCliff := offThecliff + 1;
        IF offTheCliff > SELF.fCliffThreshhold THEN
          ApplicationBeep;
      END;

    BEGIN
      offTheCliff := 0;
      myGrazerList.Each(DoGraze);
    END;
```

You use a nested procedure so that DoGraze can access the local variable offTheCliff. This allows you to use a variable that has limited scope and that is created dynamically so that you don't have to allocate a global variable. Also, since DoGraze is embedded within a TMyApplication method, you have access to the this symbol (this is the equivalent to SELF in Object Pascal).

Therefore, the problem for C++ programmers here is that there is no implicit support for getting access to local variables, such as offTheCliff, as well as the reference to the correct object through this. So what's the alternative for C++ programmers in a case like this?

First, let's take a quick look at how the Each method is declared:

Object Pascal

```
TList = OBJECT (TDynamicArray)
  ...
  PROCEDURE TList.Each(PROCEDURE DoToItem(item: TObject));
  ...
  END;
```

C++

```
class TList : public TDynamicArray {
  public:
  ...
  virtual pascal void Each(pascal void (*DoToItem)(TObject *item, void
                          *DoToItem_StaticLink), void *DoToItem_StaticLink);
  ...
}
```

As you can see, the two components of the PROCEDURE parameter have to be declared explicitly in C++. Because of this, you can come up with four different solutions to the problem, and all of them hinge on being creative with what you pass for the static link parameter.

1. Case: You need access to this, but don't need to access any local variables. Pass this in DoToItem_StaticLink directly.
2. Case: You need access to a single local variable, but not SELF. Pass the reference to that local variable in DoToItem_StaticLink.
3. Case: You need access to multiple amounts of information, including more than one local variable and this. Pass a pointer to a struct that contains this information.
4. Case: You don't need access to anything from the host procedure (including local variables and this). Pass a NIL for the static link.

Now to look at each of these in more depth.

Pass this in DoToItem_StaticLink Directly

This is the approach where you would pass this as the DoToItem_StaticLink value. You would want to do this if you needed to access your object, but didn't need to access any local variables. Here's what some C++ code would look like using this method. You pass this as the static link parameter and convert it back into an object reference in your callback procedure.

```
pascal void DoGraze(TObject* item, void* staticLink) {
    TMyApplication *self;

    self = (TMyApplication *) staticLink;
    self->DoSomethingElse();
    ((TGrazer *)item)->Graze();
}

pascal void TMyApplication::GrazeAll() {

    myGrazerList->Each(DoGraze, this);
}
```

Pass the Reference to a Single Local Variable in DoToItem_StaticLink

You would use this method if all you had to do was access a local variable of your host procedure. Getting to your local variable is now just a matter of dereferencing the staticLink parameter.

```
pascal void CountGrazers(TObject* item, void* staticLink) {
    int *grazerCountPtr = (int *) staticLink;

    ++(*grazerCountPtr);
}

pascal void TMyApplication::GrazeAll() {
    int grazerCount = 0;

    myGrazerList->Each(CountGrazers, &grazerCount);
}
```

Pass a Pointer to a struct in DoToItem_StaticLink

If you need to pass multiple amounts of information, such as more than one local variable, possibly including a reference to `this`, you can do so with a `struct`. This `struct` would hold all the local variables you need to pass to the callback routine. You would declare an instance of this `struct` in your local parameter list and pass a pointer to it as the static link. In your callback procedure, you would coerce the `staticLink` variable back into a `Pointer` to this `struct`, and then get all the information you need.

An example of this could look as follows:

```
typedef struct {
    int offTheCliff;
    TMyApplication *self;
} localVars;

pascal void DoGraze(TObject* item, void* staticLink) {
    localVars *hostLocals = (localVars *) staticLink;

    ((TGrazer *)item)->Graze();
    if ((TGrazer *)item->GrazedOffTheCliff()) {
        ++(hostLocals->offTheCliff);
    }
    if (hostLocals->offTheCliff > hostLocals->self->fCliffThreshhold) {
        ApplicationBeep();
    }
}

pascal void TMyApplication::GrazeAll() {
    localVars myLocals;

    myLocals.self = this;
    myLocals.offTheCliff = 0;
    myGrazerList->Each(DoGraze, &myLocals);
}
```

Pass a Zero for the Static Link

You would do this in situations where you can get by with the formal parameters that are given to you and don't need to access any of your host's local variables or the object reference. Since passing a zero means "don't push a static link onto the stack" in this convention, you have to adjust the parameter list of your callback `DoGraze` accordingly.

```
typedef pascal void (* EachProcType)(TObject *, void *);

pascal void DoGraze(TObject* item) {
    ((TGrazer *)item)->Graze();
}

pascal void TMyApplication::GrazeAll() {
    myGrazerList->Each((EachProcType)DoGraze, nil);
}
```

Which of these methods you use is up to you.

Further Reference:

- *MPW 3.0 Pascal Reference*, Chapter 8, pp. 145-147
- Your dentist, twice a year

Macintosh
Technical Notes

Developer Technical Support

#266: Absolute Pointing Device Memory Structure

Revised by: Dave Fleck, Don Ko & Richard Breen — April 1992
Written by: David Stevenson — February 1990

This Technical Note specifies a memory data structure for use by absolute pointing devices; it was developed for the Apple Desktop Bus (ADB) but could also be used for devices using serial input. Generally, this data structure is created and updated by the pointing device's driver and read by either an application or the system cursor rendering software.

Definitions

The data structure should reside in memory allocated by the driver at installation time.

An application should attempt to access the data structure via an OpenDriver call, passing a name of ".APD" (Absolute Pointing Device). If the call is successful, a Status call should then be issued, passing a code of 20 and a pointer to a longword where the address of the data structure will be returned.

If the OpenDriver call fails, a GetADBInfo call should be issued as described below.

For an ADB pointing device driver, this structure should be pointed to by the "optional data area" `Pointer` which is passed to `_SetADBInfo`. System software and application software can then find this data structure by calling `_GetADBInfo` with an ADB device address of 4 and dereferencing the data area pointer.

Pointing devices and drivers that support this data structure should indicate that they do so by providing the following identification: the ASCII characters `'TBLT'` in the tenth longword of the header and the number of the particular version of the structure the driver supports in the version field of the first longword of the header. Figure 1 shows the format of a longword.

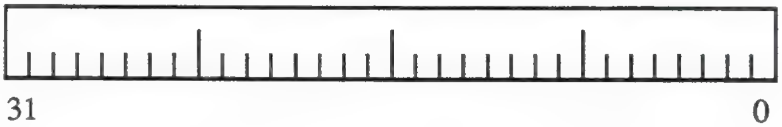


Figure 1 Format of a Longword

Bit fields within a longword are specified by a-b where a is the leftmost bit and b is the rightmost bit; for example, 7–0 specifies the fourth least significant byte in the longword.

The data structure for tablets and other absolute pointing devices consists of a 40-byte header plus one 60-byte block per cursor. The header contains a pointer that can be used to extend this structure to contain additional information supplied by particular devices.

Header

The first longword of the header consists of the following, as illustrated in Figure 2: data structure version number, synchronization semaphore, number of cursors, update flags.

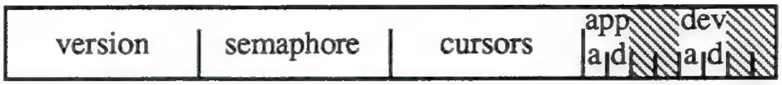


Figure 2 Format of Header's First Longword

An eight-bit field (version, bits 31–24) indicates the version number of the data structure format: a value of all zeros or all ones is invalid and indicates an uninitialized or possibly corrupted data structure; a value of 2 indicates that the format is the one here. The second eight-bit field (semaphore, bits 23–16) is used as a semaphore to coordinate access to this data structure among multiple processes. (There is currently no operating system support for multiple processes synchronizing by using a semaphore.) The third eight-bit field (cursors, bits 15–8) indicates the number of cursors on the device. Two pairs of flags are used to indicate updated information (new attributes (a) or new data (d)): one pair (app a, bit 7, and app d, bit 6) is set by the application and read and cleared by the device driver, while the other pair (dev a, bit 3, and dev d, bit 2) is set by the driver and read and cleared by the application. Other bits in the fourth byte (bits 5–4 and 1–0) must be zero.

The second longword of the header contains the resolution.

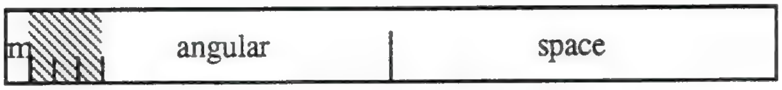


Figure 3 Format of Header's Second Longword

An English and metric flag (m, bit 31) indicates whether measurements are in English (the bit is a 0) or metric (the bit is a 1) units. Other bits in the first byte (bits 30–28) must be zero. A 12-bit field (angular, bits 27–16) indicates the angular resolution sensed by the pointing device; the value in this field indicates the number of angular inclinations that can be sensed; the application can map this range into a full 360°, 180°, or actual inclination range of the device (this field is meaningful only if the orientation format reports angular measure, that is, if tilt or an attitude matrix is supported). A 16-bit field (space, bits 15–0) specifies the spatial resolution of the device (the units of measure used to specify the dimension of the sense area and origin offset (displacement) longwords in the header and in the coordinate longwords in the cursor blocks: units per inch for English and units per centimeter for metric). Thus, for example, 1000 in this field would indicate that dimensions are specified in 0.001 inches.

The third longword of the header contains the X axis dimension of the sense area, the fourth longword the Y axis dimension of the sense area, and the fifth longword the Z axis dimension of the sense area. (If a tablet supports both portrait and landscape modes, this is reflected in the values in the X and Y dimensions.)

The sixth longword of the header contains the X axis displacement of the sense area, the seventh longword the Y axis displacement of the sense area, and the eighth longword the Z axis displacement of the sense area. These axis displacements specify the X, Y, and Z coordinate values, respectively, at the sense area origin. The sense area origin is at the lower left in a 2-D sense area such as a tablet. The dimensions of the sense area are from a reference point; the displacements are the minimum values the device returns.

The ninth longword of the header contains a pointer to the first byte of a block that specifies device-specific extensions. For example, if the device is intelligent, this may point to an area that contains a command buffer for programming the device.

The tenth longword of the header contains four ASCII characters identifying the device type, in this case `'TBLT'`.

Cursor Block

For each cursor, the following block of information is provided.

The first longword contains a transducer type, capability flags, pressure resolution, and orientation format, as illustrated in Figure 4.

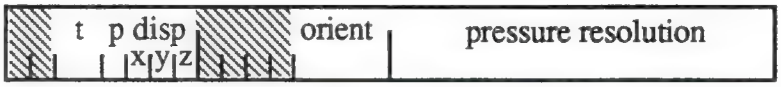


Figure 4 Format of Cursor Block's First Longword

The two-bit transducer type field (t, bits 29–28) specifies the pointing device: zero indicates that the type is unknown, one indicates pen, and two indicates a cursor. Three flags (disp x, disp y, and disp z; bits 26, 25, and 24) specify the capability to sense displacements in the X, Y, and Z dimensions, with a one indicating the capability is present. A four-bit field (orient, bits 19–16) indicates the format of orientation information: 0 indicates no orientation information is provided, 1 indicates an attitude matrix is supplied, 2 indicates pen tilt from vertical, and 23–20 are reserved for additional formats. A flag (p, bit 27) indicates whether tangential pressure is sensed, with a one indicating pressure is sensed. A 16-bit field (pressure resolution) indicates the number of pressure levels that can be sensed; zero in this field means that pressure is not sensed. This number applies to both normal and tangential pressure.

As illustrated in Figures 5 and 6, the second and third longwords of the cursor block contain, respectively, the x scale and translation factors and y scale and translation factors to use when mapping between a screen window and an area on the tablet. The scaling and translation factors are provided by the driver. (Typically the factors are calculated from the tablet to screen mapping set by the user.) An application can read these factors or it can save and replace them with its own. The saved factors should be restored when the application terminates.

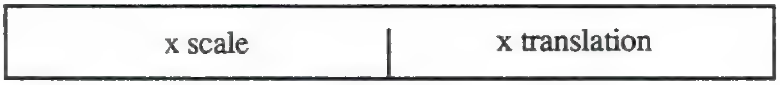


Figure 5 Format of Cursor Block's Second Longword

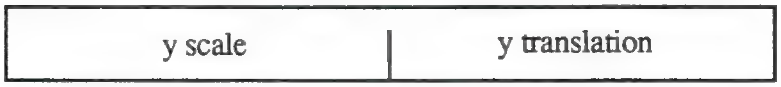


Figure 6 Format of Cursor Block's Third Longword

The x and y scale factors are positive fractions (related to the zoom factor of the application) and the translation values are such that the relation between the (mouse) cursor position and the tablet coordinates are as follows:

```
mouse x = (tablet horizontal) * (x scale) + (x translation)
mouse y = -(tablet vertical) * (y scale) + (y translation)
```

The negative in the y equation compensates for the different orientation of the tablet coordinate origin and the screen coordinate origin. (Tablet horizontal is normally the x coordinate and tablet vertical the y coordinate. See Note 1 for an example calculation.)

The fourth longword of the cursor block contains the proximity flag, cursor update flag, mouse event flag, number of buttons, mouse-down pressure threshold, and button mask, as illustrated in Figure 7.

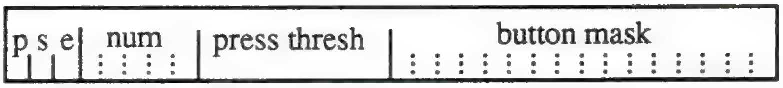


Figure 7 Format of Cursor Block's Fourth Longword

The mouse event bit (e, bit 29) is set to 1 by the driver if any button that provides a mouse click function is pressed. This bit is valid only when the s bit is 0. A 1 indicates that a mouse button is "down"; a 0 indicates all mouse buttons are "up."

The mouse-down pressure threshold (bits 23–16) applies to a transducer capable of sensing pressure at one or more of its switches (for example, a stylus with a pressure-sensitive tip). The mouse-down pressure threshold value is set by the user (typically via a CDEV). It specifies the pressure level that must be met or exceeded for a mouse-down to be posted by the driver. When the pressure level drops below the threshold, the driver will post a mouse-up. If the pressure resolution (longword 1, lower word) is 256 or less, the threshold value can be used without modification. If the resolution exceeds 256, the threshold should be scaled as follows:

```
scaled threshold = threshold * (pressure resolution / 256)
```

The proximity flag (p, bit 31) indicates whether the cursor is within the sense area (or was in proximity the last time the data structure was updated), with a one indicating that the cursor is in proximity. The cursor update flag (s, bit 30) is used to disable the driver from updating the system cursor or from posting mouse-up and mouse-down events; when this flag is set, the application assumes this responsibility. A five-bit field (num, bits 28–24) specifies the number of buttons on the device; a value of zero in this field indicates that the number of buttons is unknown. The button mask (button mask, bits 15–0 corresponding to buttons 15–0) indicates which buttons are used by the driver (for example, for system cursor control and mouse-up and mouse-down events) and which buttons are available for application usage when the s bit is zero. A value of 0 indicates that the button is being used by the driver; a value of 1 indicates that the button is available for use by an application. When the s bit is 1, the activity of all buttons is reported in the button update field. When the s bit is 0, only the activity of buttons marked as available (value of 1) in the button mask is reported. The button mask field is intended primarily to be read by an application. An application may alter it, but should restore it upon termination. See Note 2 for more details.

The fifth longword of the cursor block contains the button update. The format of this longword is illustrated in Figure 8.

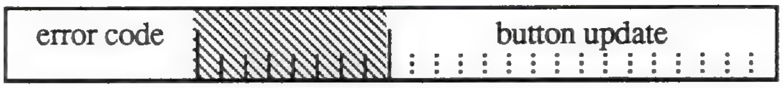


Figure 8 Format of Cursor Block's Fifth Longword

An eight-bit field (error, bits 31–16) can be used for an error code return; a zero in this field indicates no error (or no error code is returned). A 16-bit field (button update, bits 15–0 corresponding to buttons 15–0) encodes the active switch or switches, one switch per bit, with a one in the bit position indicating that the button is active. By convention, for a stylus, bit zero is identified with the tip.

The sixth longword of the cursor block contains the pressure. Two 16-bit unsigned numbers (right justified) indicate detected pressure in normal and tangential directions. Figure 9 illustrates the format of this longword.

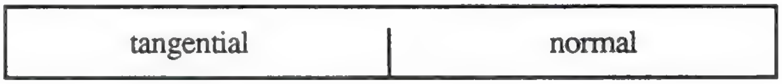


Figure 9 Format of Cursor Block's Sixth Longword

The seventh longword of the cursor block contains the time stamp. Each time this block of information is updated, this field records the number of ticks since system startup.

The eighth, ninth, and tenth longwords of the cursor block contain the X, Y, and Z coordinates respectively.

The eleventh through fifteenth longwords contain orientation information. For an attitude matrix, each longword contains two 16-bit fields; nine elements can be used to construct the attitude matrix. A_{11} and A_{12} are in the first longword, A_{13} and A_{21} are in the second, A_{22} and A_{23} are in the third, A_{31} and A_{32} are in the fourth, and A_{33} is in the fifth, as illustrated in Figure 10.

A_{11}	A_{12}
A_{13}	A_{21}
A_{22}	A_{23}
A_{31}	A_{32}
A_{33}	▨

Figure 10 Attitude Matrix

For pen tilt, the x-tilt is in the upper 16 bits of longword nine, while the y-tilt is in the lower 16 bits; the other four longwords are unused as illustrated in Figure 11.

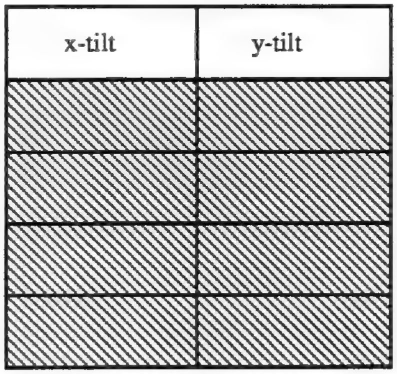


Figure 11 X-Tilt and Y-Tilt

The tilt values are signed numbers. The tilt values range from –(angular resolution / 2) to +(angular resolution / 2). When the pen is perpendicular to the tablet in both the x and y coordinates, the x and y tilt values are 0.

Notes

1. The following is an example of mapping between a screen window and the tablet:

   Assume that an area on the tablet with absolute coordinates (0,0), (0,10000), (10000,10000), (10000,0) is to be mapped onto a screen window with corresponding screen coordinates (0,720), (0,0), (720,0), (720,720); that is, the tablet's origin is in the bottom left corner while the window's is in the top left, and the tablet is 10,000 unit square while the window is 720 unit square. The x scale is 720/10000=0.072, as is the y scale. The value would actually be calculated using a 32 by 16 divide, with the numerator (720) in the upper 16 bits of the 32-bit numerator (and 0 in the lower 16 bits); the result is the "fractional" part of the division to be stored in the scale field in the memory structure. The x translation is 0 and the y translation is 720. These figures can be determined by taking the known coordinates for a single point in both frames and solving each equation for the following translation, using the coordinates for the tablet's absolute origin:

   ```
   0 = mouse x = (0)*(.072) + x translation = 0
   720 = mouse y = -(10000)*(.072) + y translation = 0
   ```

   Now the tablet coordinate (1098, 253) is mapped into the window coordinate system (79,702):

   ```
   mouse x = (1098)*(.072) + 0 = 79
   mouse y = -(253)*(.072) + 720 = 702
   ```

2. An application, Control Panel device, or other (non-driver) entity may modify the following fields:

   Header, word 1: semaphore, app and dev flags;

Cursor Block, words 2,3: scaling and translation, and word 4: s bit, mouse-down pressure threshold (only if deemed necessary), and button mask.

In particular, the English or metric flag in Header word 2 is established by the driver and cannot be changed by an application, Control Panel device, or other non-driver entity.

Setting the s bit to 1 allows the application to view the state of all buttons via the button update field. It inhibits the driver, however, from performing any user defined functions associated with the buttons. If an application wishes to use some of the buttons, but leave some available for user functions, it can change the button mask (leaving the s bit at 0). The application should save the mask before replacing it with its own settings. Mouse events may or may not continue to be received depending on whether the application has taken control of the mouse function button(s). Upon termination the application should restore the button mask.

The mouse-down pressure threshold is basically intended to be a read-only field. It indicates the user's preference for the amount of pressure to be exerted for a mouse-down to occur. Although it's not advised, an application can overwrite this field with its own value and restore it upon termination.

3. When an application terminates it should restore any parameters it may have changed.

Further Reference:

- *Inside Macintosh*, Volume V, The Apple Desktop Bus
- Technical Note #206, Space Aliens Ate My Mouse (ADB—The Untold Story)
- MacDTS Sample Code #17, TbltDrvr

Macintosh
Technical Notes

Developer Technical Support

#267: TextEdit Technicalities

Revised by: Mary Burke April 1990
Written by: Mary Burke February 1990

This Technical Note discusses some areas in TextEdit that have not previously been clearly documented.
Changes since February 1990: Added a note about the changes in TextEdit for System Software 6.0.5, documented the low-memory global `TESysJust`, clarified information about text direction and `_TESetJust`, discussed problems with the `SetWordBreak` routine along with a solution to work around it, and described the differences in dialog text item behavior.

TextEdit in 6.0.5

In addition to all the features of earlier versions, TextEdit 3.0 now allows you to take advantage of the Script Manager's handling of systems with more than one script system installed. TextEdit uses the Script Manager to support such systems and now exhibits the correct behavior for editing and displaying text in multiple styles and different scripts. Multiple scripts can even exist on a single line due to TextEdit's use of the Script Manager. The new version of TextEdit in 6.0.5:

- handles mixed-directional text
- synchronizes keyboards and fonts
- handles double-byte characters
- determines word boundaries and line breaks
- provides outline highlighting in the background
- buffers text for performance improvements
- permits left justification in right-to-left directional scripts
- customizes word breaking
- customizes measuring

Refer to the TextEdit chapter in *Inside Macintosh*, Volume VI, for detailed documentation on TextEdit 3.0. If you do not have Inside Macintosh, Volume VI, contact Developer Technical Support at the address listed in Technical Note #0 for a copy of this documentation.

The LineStarts Array and nLines

The LineStarts array is a field in a TextEdit record that contains the offset position of the first character of **each** line. This array has the following boundary conditions:

- It is a zero-based array.
- The last entry in the array must have the same value as teLength.
- The maximum number of entries is 16,000.

To determine the length of a line you can use the information contained in the lineStarts array and nLines. For example, if you want to determine the length of line n, subtract the value contained in entry n of the array from the value in the entry (n+1):

```
    lengthOfLineN := myTE^^.lineStarts[n+1] - myTE^^.lineStarts[n];
```

The terminating condition for this measurement is when n = nLines + 1. It is important not to change the information contained in the array.

TESysJust

TESysJust is a low-memory global that specifies the system justification. The default value of this global is normally based on the system script. It is -1 when a system's default line direction is right to left, and 0 for a default left-to-right line direction. Applications may change the value using the Script Manager routine SetSysJust; however, these applications should save the current value before using it and restore it before exiting the application or processing a MultiFinder suspend event. The current value may be obtained using the Script Manager routine GetSysJust.

Forcing Text Direction

The original TextEdit documentation introduced _TESetJust with three possible choices for justification: teJustLeft (0), teJustCenter (1), and teJustRight (-1). These choices are appropriate for script systems that are read from left to right. However, in script systems that are read from right to left, text is incorrectly displayed as left justified in dialog boxes and in other areas of applications where users cannot explicitly set the justification. To fix this problem, the behavior of teJustLeft has changed to match the line direction of the system in use, which is the value stored in TESysJust. Another constant has been added to allow an application to force left justification: teForceLeft (-2). This constant has been available for some time, but it has not been documented until now. If your application does not allow the user to change the justification, then it should use teJustLeft; if it does, then it should use teForceLeft for left justification.

A Little More on Redraw in _TESetStyle

If the redraw parameter used in _TESetStyle is FALSE, line breaks, line heights, and line ascents are not recalculated. Therefore a succeeding call to a routine using any of this information does not reflect the new style information. For example, a call to _TEGetHeight (which returns a total height between two specified lines) uses the line height set previous to the _TESetStyle call. A call to _TECalText is necessary to update this information. If redraw is TRUE, the current style information is reflected. This behavior also holds for the redraw parameter in _TEReplaceStyle.

TEDispatchRec

There is currently space reserved for four documented hooks in the TEDispatchRec: TEEolHook, TEWidthHook, TEDrawHook and TEHitTestHook. The space beyond these hooks is reserved, and any attempt to use this private area results in corrupted TextEdit data.

Custom Word Breaks

A problem exists in one of TextEdit's advanced procedures, SetWordBreak. The current glue code does not preserve the state of the registers correctly; however, the solution is fairly simple. Instead of calling SetWordBreak and passing a pointer to your custom word break routine, pass the pointer to your external glue which should call your custom word break routine. Following is the glue code that correctly handles the registers:

```
WordBreakProc    PROC       EXPORT

                 IMPORT     MYWORDBREAK        ;Must be uppercase here
                 MOVEM.L    D1-D2/A1,-(SP)
                 CLR.W      -(SP)              ;Space for result
                 MOVE.L     A0,-(SP)           ;Move the ptr to stack
                 MOVE.W     D0,-(SP)           ;Move the charpos to Stack
                 JSR        MYWORDBREAK
                 MOVE.W     (SP)+,D0           ;Set Z bit
                 MOVEM.L    (SP)+,D1-D2/A1
                 RTS

                 ENDP
```

An external declaration is also necessary:

```
    FUNCTION WordBreakProc( text: Ptr; charPos: INTEGER ) : BOOLEAN; EXTERNAL;
```

as is the function itself. One thing that should be noted is that it is not really necessary to have MyWordBreak boolean, but rather to have the Z bit set properly. The result of the function should be zero when you do not want a break; otherwise, a non-zero value indicates a break is desired.

```
    FUNCTION MyWordBreak( text : Ptr; charPos : INTEGER ) : INTEGER;
    { Your word break code here. }
```

For more information, refer to the TextEdit chapter of *Inside Macintosh*, Volume I-380.

Static and Editable Text

The Dialog Manager depends on TextEdit to display text in dialog boxes. For an editable text field, the Dialog Manager simply calls _TEUpdate. Before making this call, it may double the width of the rectangle to contain the text if the height of the rectangle is sufficient for only one line and the line direction specified by TESysJust is left to right. In this case, the Dialog Manager extends the rectangle on the right. Note, however, this does not occur when your line direction is right to left.

For static text items, _TextBox is used instead. When the display rectangle is not large enough. _TextBox clips the text to the size of the specified rectangle. To avoid the clipping problem, simply make the display rectangle larger. If your dialog box contains both static and editable text items, the difference in the text handling may appear inconsistent.

Further Reference:

- *Inside Macintosh*, Volumes I,V & VI, TextEdit
- *Inside Macintosh*, Volume V, The Script Manager
- *Inside Macintosh,* Volume I, The Dialog Manager
- Technical Note #207, TextEdit Changes in System 6.0

#268: MacinTalk—The Final Chapter

Written by: Jim Reekes

February 1990

This Technical Note discusses the MacinTalk software product.

The Introduction

For the introduction of the Macintosh computer, it was decided (by the powers formerly in charge) that such a computer would need something very special to make it a unique event. To aid in this concept, a third-party company was contracted to write a speech synthesizer which would allow the Macintosh computer to introduce itself. The contract was signed, and the work begun.

The outcome of this work was MacinTalk. MacinTalk is a file that can be placed into the System Folder of an ordinary Macintosh computer and allow text to be transformed into speech for the introduction in 1984. It was felt to be an interesting piece of software, so Apple made it available to developers. Interfaces to MacinTalk were published and Apple Software Licensing allowed it to be included with developers' products.

The original project was to get a speech driver for the Macintosh, but it did not include obtaining the source code to this driver. Apple only has exactly what it gives to developers: a file to be copied into the System Folder, and this file cannot be changed since Apple does not have the source code.

MacinTalk works by using a VBL task to write data directly to the sound hardware of the Macintosh Plus and SE logic boards—a method which Apple does not support. It has only been through the efforts of the Sound Manager that software that writes directly to this sound hardware continues to work. MacinTalk continues to write to the hardware addresses of the Macintosh 128K logic board, but the Sound Manager and the Apple Sound Chip work together to allow programs like MacinTalk to continue working on newer machines. The Sound Manager and the Apple Sound Chip were introduced with the Macintosh II.

The Sound Manager watches the hardware addresses that used to be present on the Macintosh. When the Sound Manager detects activity at one of these addresses, it goes into a "compatibility" mode. In this mode, it routes the data to the real sound hardware, but while this is happening, proper Sound Manager code cannot run—even the Sound Manager's `_SysBeep` does not work when MacinTalk is in use. Furthermore, the compatibility mode cannot be turned off until the application requiring it calls `_ExitToShell`. Even an application that uses sound properly, with correct code, does not work if another application opens the MacinTalk driver. There are **no** solutions to this incompatibility. MultiFinder and System 7.0 allow more than one application to run concurrently, so it is possible for an older application which uses MacinTalk to limit the features and abilities of a new and improved application.

Around the time of the Macintosh II introduction, Apple made a single hack to MacinTalk to remove self-modifying code which caused a problem for 68020-based machines. This modified version, known as 1.31, was available from APDA with the warning that "Apple does not provide any support for MacinTalk. Not even to Apple Certified Developers." While this warning is still true and APDA continues to sell MacinTalk 1.31 as a "Class 3" product, the official Apple position about it is: "Apple Computer, Inc. does not recommend that you use Class 3 products for developing commercial software—they are intended for your personal enjoyment only."

In other words, if you find MacinTalk interesting and entertaining—go ahead and purchase it. Write some code and enjoy. However, be warned that MacinTalk should **not** be included as part of **any** commercial product. Apple Computer, Inc. provides no support for MacinTalk other than what is purchased with the package itself, and there will be no support in the future. Apple is committed to providing the developer community with an array of speech technologies integrated with the Sound Manager. In preparation for certain system software changes, such as System 7.0, Apple does not recommend the continued use of the existing MacinTalk package.

The Surgeon General Warning

The development of MacinTalk ended with the introduction of the Macintosh in 1984. Nothing more will be done to this product. It is a compatibility risk to use MacinTalk. It causes the Sound Manager to fail. It will **not** work with the new Sound Manager planned for System 7.0. This system revision will also introduce the new feature of virtual memory; it is not expected that MacinTalk will work while the user has Virtual Memory running. It may not work at all with future versions of the Macintosh hardware. Continued use of MacinTalk is a major compatibility risk. Do not operate heavy machinery while under the influence of this product.

Further Reference:

- *Inside Macintosh*, Volume V, Compatibility Guidelines
- *The Sound Manager*, Interim Chapter (DTS)
- Technical Note #19, How to Produce Continuous Sound Without Clicking

Macintosh
Technical Notes

Developer Technical Support

#269: 'ckid' Resource Format

Written by: Keith Rollin April 1990

This Technical Note describes the `'ckid'` resource format used by MPW's Projector. If you are writing an editor or development system, you may wish to allow or disallow file modification based on the information in the resource.

MPW 3.0 and greater implement a source code control system called Projector. Projector manages sets of source code files known as projects. Users are able to check out source code, make modifications to it, and check it back into the project. Source code can be checked out as "modifiable" or "read-only." Only one modifiable version of a source file is allowed to be checked out at a time. This feature is very useful if you store your files on a server such as AppleShare; anyone else trying to check out the same file can only check it out "read-only," ensuring that two people are not modifying the same file at the same time.

Note: This is an overly simplistic description of Projector. It can actually manage more than just source code and has provisions for variations on the modifiable or read-only scheme, such as being able to create experimental offshoots of the main source code called branches.

MPW attaches Projector information to checked out files by adding `'ckid'` resources to them. This resource contains information such as to what project the file is attached, when the file was checked out, who checked it out, and whether or not it was checked out read-only. If it **is** checked out read-only, the MPW Shell takes note of this, and does not allow the user to edit the file.

If you are working on a development or source code editing system, then you may wish to respect the `'ckid'` resource. At the very least, this means not deleting it. When saving changes to an existing file, some applications write the modified data to a temporary file, delete the old file, and rename the temporary file to have the same name as the original. Unfortunately, this deletes the `'ckid'` resource, preventing the user from checking their file back into the Projector database. Applications saving files with this technique should transfer the resource over before deleting the old file.

If you want to include more support for the resource, you could prevent the user from making changes if the file is checked out read-only. This can be done by reading the `'ckid'` resource and looking at the appropriate fields. The entire format of the `'ckid'` resource is appended to the end of this Note. You should be interested in only four fields of the resource:

`version` This two-byte field holds the version number of the `'ckid'` resource. The current version number is 4. The information presented in this Note is valid for this version number **only**. Any attempt to apply the information presented here to a `'ckid'` resource with a different version would be bad.

`checkSum` This four-byte field holds a checksum to validate the rest of the resource. It is generated by summing all the subsequent longwords in the resource handle, skipping the checksum field itself and any extra bytes at the end that don't compose a longword.

`readOnly` This two-byte field indicates whether the attached file is checked out for modifications or not. If the file is not checked out for modifications, this field contains a zero. If the file is modifiable, this field is non-zero and contains special version information for Projector.

`modifyReadOnly` This one-byte field provides a limited override to the `readOnly` field. Sometimes it is desirable to be able to modify a file that has been checked out read-only. One may want to do this if they have a file checked out read-only, but later decide to make modifications to it and no longer have access to the Projector database to check out a modifiable version. Under MPW, the user can execute the `ModifyReadOnly` command. This sets the `modifyReadOnly` field to non-zero, indicating that the file can be edited, even though it is checked out read-only.

In your application, you may wish to inhibit modifications to a file if it has a `'ckid'` resource and has been checked out read-only. In addition, as a convenience to your customers, you may wish to include a ModifyReadOnly feature of your own. To do this, you would need to set the `modifyReadOnly` field to non-zero and recalculate the checksum.

The following routines can help perform these functions. `CKIDIsModifiable` takes a handle to a `'ckid'` resource and returns `TRUE` if it indicates that the file is modifiable and `FALSE` otherwise. `HandleCheckSum` takes a handle to a `'ckid'` resource and returns a calculated checksum.

MPW Pascal

```
TYPE
  CKIDRec          = PACKED RECORD
    checkSum:        LONGINT;
    LOC:             LONGINT;
    version:         INTEGER;
    readOnly:        INTEGER;
    branch:          BYTE;
    modifyReadOnly: Boolean;
    { There's more, but this is all we need }
    END;
  CKIDPtr          = ^CKIDRec;
  CKIDHandle       = ^CKIDPtr;

FUNCTION CKIDIsModifiable(ckid: CKIDHandle): Boolean;

  BEGIN
    IF ckid = NIL THEN
      CKIDIsModifiable := TRUE
    ELSE
      WITH ckid^^ DO
        CKIDIsModifiable := (readOnly <> 0) |
                            ((readOnly = 0) & modifyReadOnly);
  END;
```

```
FUNCTION HandleCheckSum(h: Handle): LONGINT;

  VAR
    sum:             LONGINT;
    size:            LONGINT;
    p:               LongintPtr;

  BEGIN
    sum := 0;

    size := (GetHandleSize(h) DIV SizeOf(LONGINT)) - 1;
    p := LongintPtr(h^);
    p := LongintPtr(ORD(p) + SizeOf(LONGINT)); { skip over first long
                                                (checksum field) }
    WHILE (size > 0) DO BEGIN
      size := size - 1;
      sum := sum + p^;
      p := LongintPtr(ORD(p) + SizeOf(LONGINT));
    END;

    HandleCheckSum := sum;
  END;
```

MPW C

```
typedef unsigned long uLong;

typedef struct {
  uLong      checkSum;
  long       LOC;
  short      version;
  short      readOnly;
  char       branch;
  Boolean    modifyReadOnly;
  /* There's more, but this is all we need */
} CKIDRec, *CKIDPtr, **CKIDHandle;

pascal Boolean CKIDIsModifiable(CKIDHandle ckid)
{
  if (ckid == nil)
    return(true);
  else
    return( ((**ckid).readOnly != 0) ||
           (((**ckid).readOnly == 0) && (**ckid).modifyReadOnly));
}

pascal uLong HandleCheckSum(Handle h)
{
  long       size;
  uLong      sum = 0;
  uLong      *p;

  size = (GetHandleSize(h) / sizeof(long)) - 1;
  p = (uLong *) *h;
  p++;                    /* skip over first long (checksum field) */
  while (size-- > 0) {
    sum += *p++;
  }

  return(sum);
}
```

If you wanted to include a ModifyReadOnly function, you could use something like the following Pascal fragment:

```
h := CKIDHandle(Get1Resource('ckid', 128));
IF (h <> NIL) & (h^^.version = 4) THEN BEGIN
  h^^.modifyReadOnly := TRUE;
  h^^.checkSum := HandleCheckSum(Handle(h));
  ChangedResource(Handle(h));
END;
```

'ckid' Resource format

This MPW Rez resource template is for your application's **information** only. It is valid **only** for version 4 of the resource. Please do not write to this resource or create one of your own. If you feel that you need to change fields in the resource, then limit yourself to the `checkSum` and `modifyReadOnly` fields, and **only** if the `version` field is equal to 4. This resource format **will** change in the future.

```
type 'ckid'
   {
   unsigned longint;         /* checkSum */
   unsigned longint LOC = 1071985200;  /* location identifier */
   integer version = 4;      /* ckid version number */
   integer readOnly = 0;     /* Check out state, if = 0 it is modifiable */
   Byte noBranch = 0;        /* if modifiable & Byte != 0 then branch was made
                                on check out */
   Byte clean = 0,
        MODIFIED = 1;        /* did user execute "ModifyReadOnly" on this file? */
   unsigned longint UNUSED;  /* not used */
   unsigned longint;         /* date and time of checkout */
   unsigned longint;         /* mod date of file */

   unsigned longint;         /* PID.a */
   unsigned longint;         /* PID.b */

   integer;                  /* user ID */
   integer;                  /* file ID */
   integer;                  /* rev ID */

   pstring;                  /* Project path */
   Byte = 0;
   pstring;                  /* User name */
   Byte = 0;
   pstring;                  /* Revision number */
   Byte = 0;
   pstring;                  /* File name */
   Byte = 0;
   pstring;                  /* task */
   Byte = 0;
   wstring;                  /* comment */
   Byte = 0;
};
```

Notes

The `branch` field (field 5) holds the letter of this branch (i.e., “a”, “b”, “c”, etc.). It holds zero if this revision is on the main branch.

`PID` is the Project ID. It is generated using a combination of the tick count and time on your computer in a way that should be sufficient to generate unique Project IDs for every project ever created.

The `pstring` and `wstring` fields are variable length fields (`pstring` is a string preceded by a length `BYTE`, while `wstring` is a string preceded by a length `WORD`), which means that you cannot directly represent this resource with a `RECORD` in Pascal or `struct` in C.

Further Reference:

- *Macintosh Programmer’s WorkShop 3.0 Reference*, Chapter 7, Projector: Project Management

Macintosh
Technical Notes

Developer Technical Support

#270: AppleTalk Timers Explained

Written by: Sriram Subramanian & Pete Helme April 1990

This Technical Note explains how to effectively use timers and retry mechanisms of the various AppleTalk protocols to achieve maximum performance on an internet.

The most fundamental service in an AppleTalk internet is the Data Delivery Protocol (DDP), which provides a best-effort, connectionless, packet delivery system. A sequence of packets sent using DDP on an AppleTalk internet between a pair of machines may traverse a single high-speed Ethernet network or it may wind across multiple intermediate data links such as LocalTalk, TokenRing, etc., which are connected by routers. Some packet loss is always inevitable because of the loosely coupled nature of the underlying networks. Even on a single high-speed Ethernet network, packets can be lost due to collisions or a busy destination node. The AppleTalk Transaction Protocol (ATP), the AppleTalk Data Stream Protocol (ADSP), and other high-level protocols protect again packet loss and ensure reliability by using positive acknowledgement with packet retransmission mechanism.

The basic transaction process in ATP consists of a client in a requesting node sending a Transaction Request (TReq) packet to a client in a responding node. The client in the responding node is expected to service the request and generate a Transaction Response (TResp) packet, which also serves as an acknowledgement. The ATP process in the requesting node also starts a timer when it sends a packet and retransmits a packet if the timer expires before a response arrives. In a large internet with multiple gateways, it is impossible to know how quickly acknowledgements may return to the requestor. If you set the retry time to be too small, you may be retransmitting a request while a delayed response is *en route*, but if you wait too long to retransmit a request, application performance may suffer. More importantly, the delay at each gateway depends upon the traffic, so the time required to transmit a packet and receive an acknowledgement varies from one instant to another. To further complicate matters, two packets sent back to back could take completely different routes to the destination.

Selecting ATP Retry Time And Retry Count

You can use the round trip time for a transaction as a heuristic for setting the retry time and retry count. The round trip time between two nodes in a particular internet at a particular time is usually deterministic.

The easiest way to set the retry time is to assign a static value based on the round trip time for a transaction. The AppleTalk Echo Protocol (AEP) can be used to obtain the round trip time in a given moment between two nodes. AEP is implemented in each node as a DDP client residing on statically-assigned socket number four. You should use DDP to send AEP requests through any socket that is available. You can listen for AEP responses by implementing a socket listener. The following code is an example AEP socket listener.

```
;________________________________________________________________________
;________________________________________________________________________
;
; EchoDude
;
; 3/90 pvh - MacDTS
;
; ©1990 Apple Computer, Inc.
;
; The following MPW Asm code is a socket listener for reading in returned Echo
; (DDP type 4) packets.
;
; The target device was shipped a packet with a '1' in the first byte of the data area
; by way of a DDPWrite.  It was sent to socket 4, the Echoer socket.  If the target
; device has an Echoer, it will send a return packet to us of equal size except it
; will have replaced the '1' in the first byte with the value '2'.  This indicates an
; EchoReply packet.
;
; The listener itself (RcvEcho) is added with a POpenSkt (Inside Mac V-513) call by
; passing the address of the listener in the listener field of the parameter block.
;
; All we really are trying to accomplish here is to set up a notification for returned
; packets from the target Echoer.  A time (Ticks) is stuffed into a location
; our app can find (actually back into the packet buffer) and will be used to
; calculate round trips times.  We'll also save off the hop count from the packet
; header for fun too since I have nothing better to do with my time on weekends.
;
; More could be done with this listener as far as making sure that we are only
; receiving back a packet from the node we sent it to etc.... but we can't
; encompass everything in a sample.  Okay, well we could… but we have to leave
; something for you guys to do.
;
; It should be noted that careful preservation of register A5 is necessary.
; LAP requires that A5 be preserved AFTER the call to ReadRest. i.e. you
; cannot save A5 onto the stack when your socket listener is entered, call ReadRest
; and then restore A5 from the stack and exit.  Wah.  LAP requires that the address
; placed in A5 during ReadRest be there when your socket listener is exited.
; So… if you need a different A5 after the call to ReadRest make sure you restore
; it before RTS-ing back the caller.
;
;
;    Called:
;         A0,A1,D1 : Preserve until after ReadRest
;         A2 -> MPP local variables
;         A3 -> RHA after DDP header
;         A4 -> ReadPacket, 2(A4) -> ReadRest
;         A5 Useable until ReadRest
;         A6,D4-D7 : Preserve across call
;
;________________________________________________________________________

EchoSkt     EQU     4           ; Echo socket number
EP          EQU     4           ; EP DDP protocol type
EPReq       EQU     1           ; Code for echo request
EPReply     EQU     2           ; Code for echo reply
```

```
;
; Read the packet into the echo buffer
;

RcvEcho    PROC    EXPORT
                   EXPORT    our_A5 : CODE
                   EXPORT    our_Buff : CODE
                   IMPORT    GBOB:DATA

        BRA.S     checkEcho
our_A5
        DC.L      0
our_Buff
        DC.L      0
our_Hops
        DC.W      0
our_Ticks
        DC.L      0
checkEcho
        CMP.B     #EP,-(A3)            ; Make sure it's an echo packet
        BNE.S     RcvEIgnore           ; Ignore it if not
        LEA       toRHA(A2), A3        ; top of RHA
        CLR.L     D2                   ; clean up D2
        MOVE.B    lapType(A3), D2      ; lap type
        CMP.B     #longDDP, D2         ; check for long header (Type #2 packet)
        BNE.S     noHops               ; wah... no hops if short packet
        MOVE.B    lapType+1(A3), D2    ; this is the hop count byte, 1st byte in DDP header
        AND.B     #$3C, D2             ; mask to middle 4 bits of byte for hop count
                                       ;      | x | x | H | O | P | S | x | x |
        ASR.B     #2, D2               ; shift 2 bits to right
        LEA       our_Hops, A3         ; address of our storage
        MOVE.B    D2, (A3)             ; move # of hops into our storage
noHops
        MOVE.W    #DDPMaxData, D3      ; our buffer is #DDPMaxData in size
        LEA       our_Buff, A3         ; address of buffer to read packet into
        MOVE.L    (A3), A3             ; set buffer
        JSR       2(A4)                ; ReadRest of packet into buffer
        BEQ.S     RcvEchoReply         ; If no error, continue
        BRA.S     RcvEchoFail          ; dang…
RcvEIgnore
        CLR       D3                   ; Set to ignore packet
        JMP       2(A4)                ; Ignore it, ReadRest and return
        BRA.S     RcvEchoFail
RcvEchoReply
        CMP.B     #EPReply, -DDPMaxData(A3)    ; make sure it's our reply packet
                                       ; it shouldn't be anything else, but check
                                       ; just in case
        BNE.S     RcvEchoFail          ; if not our reply then blow
        MOVE.L    A5, D2               ; save dude in D2
        LEA       our_A5, A5           ; address of our A5 local storage
        MOVE.L    (A5), A5             ; make A5 our A5 for application global use
        MOVE.B    #1, GBOB(A5)         ; set flag confirming reception of
                                       ; echo reply packet
        LEA       our_Buff, A3         ; address of our local buffer storage into A3
        MOVE.L    (A3), A3             ; get saved pointer and set buffer.
        LEA       our_Hops, A5         ; address of hops local storage… notice we
                                       ; are TRASHING A5 with this!!!!!
        MOVE.W    (A5), (A3)+          ; copy in hop count to buffer
        MOVE.L    Ticks, (A3)          ; next copy in Ticks

        MOVE.L    D2, A5               ; restore dude
        RTS                            ; return to caller
RcvEchoFail
        RTS                            ; return to caller

        ENDP
```

```
setUpSktListener    PROC    EXPORT
                            IMPORT    our_A5 : CODE
                            IMPORT    our_Buff : CODE

        LEA       our_A5, A0             ; this copies
        MOVE.L    CurrentA5, (A0)        ; this copies CurrentA5 into our local
                                         ; storage for global use in the listener
        MOVE.W    #DDPMaxData, D0        ; max size of data in a packet
        _NewPtr   CLEAR
        BNE.S     setUpFailed            ; if NIL then forget it

        LEA       our_Buff, A1           ; we need to save the pointer reference
        MOVE.L    A0, (A1)               ;  in a place the listener can find it
        MOVE.L    A0, D0                 ; return value to caller
        RTS
setUpFailed
        CLR.L     D0                     ; tell caller we failed by returning nil
                                         ; (caller expecting valid ptr returned)
        RTS

        ENDP

        END
```

We now resume our regular programming…

You should typically get an AEP response packet within a few milliseconds. If there is no response for a period of time, typically about 10 seconds, you should resend your AEP request to account for a lost request or lost packets. To be really safe, you should resend your AEP request with different data to take into account the response to the first packet coming back later. The retry time could then be simply set to `k*Round_Trip_Time`, where the value of `k` depends upon the request semantics, like total data size.

This technique of statically setting the retry time is not adequate to accommodate the varying delays encountered in a internet environment at different times. You could dynamically adjust the retry time based on an adaptive retransmission algorithm that continuously monitors round trip times and adjusts its timeout parameter accordingly. To implement an adaptive algorithm, you can record the round trip time for each transaction. One common technique is to keep the average round trip time as a weighted average and use new round trip times from transactions to change the average slowly. For example, one averaging technique[1] uses a constant weighing factor, `q`, where `0 ≤ q < 1`, to weigh the oldest average against the latest round trip time:

```
    W_aver = (q * W_aver ) + (( 1 - q) * New_Round_Trip_Time)
```

Choosing a value for `q` close to 1 makes the weighted average immune to changes that last a short time. Choosing a value for `q` close to 0 makes the weighted average respond to changes in the delay very quickly.

The total time (i.e., `retry time * retry count)` before a request is concluded as failed could be anywhere from 10 seconds to a couple of minutes, depending on the type of the client application and the relative distance between the source and the destination.

[1] Douglass Comer, InterNetworking with TCP/IP.
KARN, P. and C. PARTRIDGE [August 1987], "Improving Round-Trip Time Estimates in Reliable Transport Protocols", *Proceedings of ACM SIGCOMM 1987.*

NBP Retry Counts

You cannot really use the AEP to estimate round trip times for NBP packets because you need to use NBP to determine the internet address of the node from which an echo is being sought. In this case, you have to use the type of device that you are looking for as a heuristic for setting the retry count. The LaserWriter, for example, may be busy and not respond to a LkUp packet. In such a case, you might want to do a quick lookup to return a partial list to the user like the Chooser. You could then do a longer lookup to get a more complete list of mappings. You should use a "back off" algorithm to make the subsequent lookups further apart to generate progressively less traffic. Name lookups are expensive and produce a lot of network traffic, and name confirmation is the recommended call to use when confirming mappings obtained through early bindings. Because Name lookups are expensive, you should avoid searching all the zones in the internet.

Setting TRel Timer in SendRequest

AppleTalk Phase 2 drivers allow you to set the TRel timer in `SendRequest` or `NSendRequest` calls with ATP XO (exactly once) service so as not to be locked into the pre-AppleTalk Phase 2 time of 30 seconds. You should set this timer based on the round trip time. Generally, if the round trip time is less than one second, the default TRel time setting of 30 seconds is adequate. If the round trip time is more, you can increase the TRel time proportionately.

xppTimeout and xppRetry

The two ZIP calls, `GetZoneList` and `GetLocalZones`, made on the .XPP driver contain the ATP retry interval (in seconds) and count, in the `xppTimeout` and `xppRetry` parameters. Both these functions are ATP request-response transactions between a node and a router on the network to which the requesting node is attached. The round trip is relatively short for this transaction, and you should have very small values of `xppTimeout` and `xppRetry`, typically two and three, respectively.

Further Reference:

- *Inside AppleTalk*
- *Inside Macintosh*, Volumes II & V, The AppleTalk Manager
- Technical Note #9, Will Your AppleTalk Application Support Internets?
- Technical Note #250, AppleTalk Phase 2 on the Macintosh

#271: Macintosh IIfx: The Inside Story

Written by: Rich "I See Colors" Collyer February 1990

This Technical Note addresses various areas of potential incompatibilities with the Macintosh IIfx and current software applications and provides information about some of Apple's compatibility software updates.

What's Inside

On the Macintosh IIfx, the CPU no longer handles I/O operations like floppy disk access, SCC access, and mouse events. Instead of the CPU doing all of the work, the IIfx contains a couple of separate I/O processors, Apple custom ASICs, to handle all floppy disk, mouse, and SCC I/O. With the advent of these new I/O processors (IOP), the IIfx can handle smooth cursor movement and time consuming disk operations simultaneously. These new IOPs are just an example of the new capability of this machine.

Each of the following sections talks about the changes and added functionality which makes life difficult for some types of applications. The IOPs in the IIfx cause some applications problems, and this Note shows why certain techniques no longer work and provides solutions to work around these incompatibilities where possible. A few additional sections provide information about updated System Software or peripheral software from Apple which the IIfx requires for operation.

ADB

Applications which depend upon direct access to the ADB transceiver or its VIA registers do not work with the IIfx, because the IOP which now handles ADB is not available for direct access. As in the past, the hardware is subject to change, and applications which access it directly break when new hardware is introduced. There is no solution for applications which try to directly access the ADB hardware; these applications must now use the ADB Manager or they cannot run on the IIfx and future Macintosh models.

CD-ROM Driver

To use the AppleCD SC with a Macintosh IIfx (and IIci), you must use version 3.0.1 or later of the Apple CD-ROM drivers. Earlier versions of this driver are incompatible with this hardware. You can obtain a copy of this driver from any authorized Apple dealer, the Developer CD Series, AppleLink (Developer Services: Macintosh Developer Technical Support: Peripheral Software), and the Apple FTP site on the Internet (Apple.COM under ~ftp/pub/dts/sw.license/).

EtherTalk Driver (.ENET)

To use the Apple EtherTalk card with a Macintosh IIfx, you should use version 2.0.2 of the EtherTalk driver. Earlier versions of this driver do not perform as well with this hardware. You can obtain a copy of this driver from any authorized Apple dealer, the Developer CD Series, AppleLink (Developer Services: Macintosh Developer Technical Support: Peripheral Software), and the Apple FTP site on the Internet (Apple.COM under ~ftp/pub/dts/sw.license/).

MacsBug

To use MacsBug with a Macintosh IIfx, you must use version 6.2. Earlier versions of MacsBug are incompatible with this hardware. You can obtain a copy of MacsBug 6.2 from APDA, the Developer CD Series, and AppleLink (Developer Services: Macintosh Developer Technical Support: Tools: MacsBug).

NuBus

If the Macintosh IIfx executes a Read-Modify-Write NuBus™ code sequence to a card (i.e., `TAS` or test and set) and immediately follows it with a regular cycle Read or Write, the system hangs. The solution to this problem is to execute five `NOP` instructions between the `TAS` and the next cycle. This number of `NOP` instructions should also handle future accelerations of the CPU clock, should Apple decide to further accelerate it.

SADE MultiFinder

To use SADE with a Macintosh IIfx, you must use version 6.1b9 of MultiFinder with the Set Aside feature. Earlier versions of MultiFinder are incompatible with SADE on this hardware. You can obtain a copy of MultiFinder 6.1b9 from APDA with SADE 1.1, the Developer CD Series, and AppleLink (Developer Services: Macintosh Developer Technical Support: Tools: SADE MultiFinder). Developers may **not** distribute MultiFinder 6.1bx to customers, even if licensed to distribute Apple's Macintosh System Software.

SCC

Like the processor which controls floppy disk and ADB I/O, the IIfx has another ASIC to control the SCC, but unlike the former, this processor is capable of running in a special "IOP Bypass" mode which allows direct access to the SCC.

The new SCC architecture also contains a few other differences from the previous architecture. On the IIfx, there is no longer a VIA line available for monitoring the Wait/Request signal of the SCC. Applications which depend upon this bit have no solution to this problem and are incompatible with the IIfx. In addition, on the IIfx the vSync bit (which has been available since the Macintosh SE) has moved to a new location; however, Apple is providing developers with a trap call (`_HWPriv`) which allows applications to enable or disable this bit in its new location, thereby providing a solution for applications which depend upon this bit. For more information on this trap call, see the vSync Bit section later in this Note. Technical Note #261, Cache As Cache Can, also addresses `_HWPriv`.

IIfx Serial Switch cdev

If an application requires direct access to the SCC, then you should license the IIfx Serial Switch cdev from Apple Software Licensing. The native mode of the IIfx uses a special processor to handle all SCC work, thus increasing overall machine performance by offloading this task from the CPU. However, applications must sacrifice direct SCC access for this performance gain. The IIfx Serial Switch cdev allows applications which must directly access the SCC to bypass the processor while sacrificing the increased performance.

This cdev sets a bit in parameter RAM which the IIfx checks during startup. If "Faster" mode is chosen (default), then the IIfx uses the special processor, but if "Compatibility" mode is chosen, then the IIfx lets the CPU handle SCC processing, which allows direct access. To license this cdev, contact:

Apple Software Licensing
Apple Computer, Inc.
20525 Mariani Avenue, M/S 38-I
Cupertino, CA 95014
(408) 974-4667
AppleLink: Sw.License
Internet: Sw.License@AppleLink.Apple.com

There is no way for an application to determine in which mode it is running; therefore, if the machine is in "Faster" mode and an application attempts a direct call to the SCC, the machine crashes.

Wait/Request Bit

On previous Macintosh models, there is a Wait/Request bit on the VIA1 register A for monitoring incoming serial data while the Macintosh is busy with some other operation. When the SCC receives a character, it sets this bit in the VIA, which tells the operating system that the SCC needs attention. Since the IIfx has a dedicated processor for SCC transactions, it has no need for this mechanism. Even if a machine is using the IOP Bypass mode to directly access the SCC, this line is not active, so applications which rely upon it are incompatible with the IIfx. For more information about this bit, refer to the *Guide to the Macintosh Family Hardware*, Second Edition.

vSync Bit

The `_HWPriv` ($A198, selector 7) routine enables or disables external SCC clocking. The external clock comes to the SCC through the RTxC signal, which is connected to the GPi pin on the serial port connector. This routine is used instead of writing directly to the vSync bit on VIA1 (which is not implemented on the IIfx), and it is backpatched into all previous CPUs, except the Macintosh Plus, which does not support external clocking. `_HWPriv` only works in IOP Bypass mode on the IIfx, and is documented below for your convenience:

```
Entry:
    d0.l = routine selector = 7
    a0.l =  <port number>.w <enable/disable ext clock>.w
            23-16: port number 0 = port A, 1 = port B,... for future expansion
            15-0 : 0 = internal clocking, 1 = external clocking
Exit:
    d0.l = zero if good, -1 if error
    a0.l = <port number>.w <last state of external clock>.w
```

Synchronous SCC I/O

If an application expects to make synchronous SCC I/O calls with interrupts turned off, it does not work on the IIfx, because the new IOP serial driver uses the Deferred Task Manager, which is interrupt driven. If an application tries to do something like communicate with the IOP SCC driver when interrupts are turned off, the IIfx hangs.

SCSI

The Macintosh IIfx may cause developers problems in two areas which deal with the SCSI interface. The first are the SCSI low-memory globals. A few applications rely upon undocumented low-memory globals which point to addresses in the SCSI controller chip; however, on the Macintosh IIfx, these globals now point to an entirely different area. If an application depends upon these globals, it either does nothing or crashes on the IIfx. The second problem deals with SCSI termination. For more information about SCSI termination on the Macintosh IIfx and how it differs from previous Macintosh models, refer to Technical Note #273, SCSI Termination.

In addition, although the IIfx hardware has SCSI DMA capability, the Macintosh System Software does not yet take advantage of it. Apple recommends that you wait until the Macintosh System Software implements support for the IIfx SCSI DMA to use this hardware feature.

SWIM

On the IIfx, the floppy disk controller, the SWIM, is not directly accessible; instead, the IIfx has a processor which handles all floppy drive access. This processor, an Apple custom ASIC, is not accessible to third-party developers. The I/O processing hardware is subject to change, and applications which attempt to access it directly are likely to break when new hardware is introduced.

Apple has always recommended against direct hardware access, but some applications do it anyway, and these applications now have problems with the new IIfx hardware. The most common reason these applications access the hardware directly is to move hidden information to and from the disk. As a partial solution to this problem, the IIfx includes a new version of the Sony driver which allows applications to make a control call to get raw data from the disk. For more information on this new driver and control call, refer to Technical Note #272, What Your Sony Drives For You.

Asynchronous Disk I/O

If an application expects to make asynchronous I/O calls to the Sony driver with interrupts turned off, it does not work on the IIfx, because the new IOP drivers are interrupt driven. If an application tries to do something like open a resource when interrupts are turned off, the IIfx hangs.

VIA2

All of the functionality of VIA2 has been moved to other chips in Macintosh IIfx, so if an application depends on VIA2 registers, it must find a different way to get the information for which it is looking to be compatible with the IIfx.

This is What Makes a "Wicked Fast" Macintosh

The basic message of this Note is that if developers directly access Macintosh hardware, their applications are likely to break on new hardware like the Macintosh IIfx. If an application is having compatibility problems with the IIfx, they are probably due to one of these documented changes, and this Note should help provide the necessary solutions where they are available. If an application is having compatibility problems with the IIfx and they are not related to one of these areas, then qualified developers should contact Developer Technical Support for help in tracking down the problem.

Further Reference:

- *Guide to the Macintosh Family Hardware,* Second Edition
- *Inside Macintosh*, Volume V, Compatibility Guidelines
- Inside Macintosh, Volume V, Deferred Task Manager
- Technical Note #2, Compatibility Guidelines
- Technical Note #117, Compatibility: Why and How
- Technical Note #129, _SysEnvirons: System 6.0 and Beyond
- Technical Note #261, Cache as Cache Can.
- Technical Note #272, What Your Sony Drives For You.
- Technical Note #273, SCSI Termination

NuBus is a trademark of Texas Instruments.

#272: What Your Sony Drives For You

Revised by: Rich “I See Colors” Collyer & Cameron Birse June 1990
Written by: Rich “I See Colors” Collyer & Cameron Birse April 1990

This Technical Note discusses the Sony driver control and status calls that are available on the Macintosh.
Changes since April 1990: Corrected Figure 2, since the Return Physical Drive Icon (`csCode = 21`) returns an error message instead of an icon on the Macintosh Plus.

This Note covers the external (software) interface to the Sony 3.5” floppy disk and Hard Disk 20 driver. It describes all the new calls, including those for Modified Frequency Modulation (MFM) driver versions. This discussion assumes a general understanding of the operation of Macintosh drivers. As all of these calls are not available on all Macintosh models, the following table shows which calls are available on which models:

All	`Read, Write, Kill, Eject, Set Tag Buffer, Drive Status`
128K and later ROMs	`Verify Disk, Format Disk, Track Cache Control, Return Physical Drive Icon`
256K and later ROMS	`Return Media Icon, Return Drive Info`
SuperDrive equipped	`Return Format List`
IIfx only	`Diagnostic Raw Track Dump`

Prime (Read & Write) Calls

Read and write calls to Macintosh drivers are described in general in *Inside Macintosh*, Volume II, The Device Manager, but for completeness, this discussion also includes them. The Device Manager prime routines expect to have the following fields set up in the I/O parameter block:

`ioCompletion`	pointer to a completion routine (asynchronous calls) or `NIL` (synchronous calls)
`ioVRefNum`	drive number (for device calls) or volume reference number (for file system calls)
`ioRefNum`	driver’s reference number (-5 for floppy disks or -2 for Hard Disk 20)
`ioBuffer`	pointer to the location in memory where data is read to or written from
`ioReqCount`	number of bytes to read from or write to the disk
`ioPosMode`	tells what the absolute starting point is: beginning, end, or current location (bit 6 is set to 1 to do a read-verify instead of a read)
`ioPosOffset`	offset in bytes relative to the starting point in `ioPosMode`

When you make a call to the Sony driver's prime routine, register A0 points to this I/O parameter block and register A1 points to the driver's Device Control Entry (DCE). The Device Manager sets the ioTrap field of the parameter block to either $A002 for a read request or $A003 for a write request, so the driver can determine the appropriate action. The Device Manager also sets the dCtlPosition field of the driver's DCE to the starting byte offset relative to the beginning of the disk.

You can call the Sony driver either synchronously or asynchronously; however, making an immediate "mode" call to the driver causes it to bomb. The driver begins a read or write request, returns control to the caller (either the user (asynchronous) or the Device Manager (synchronous)), then completes the request asynchronously at the interrupt level. When the request is completed or aborted, the driver returns one of the following result codes:

noErr	0	no error
wPrErr	-44	diskette is write protected
paramErr	-50	some of the requested blocks are past the end of the disk or ioReqCount is not an even multiple of 512 bytes
nsDrvErr	-56	no such drive number
noDriveErr	-64	drive not installed
offLinErr	-65	read or write request made to an ejected disk
noNybErr	-66	could not find five nibbles in 200 tries (Group Coded Recording (GCR)) or byte timeout (MFM)
noAdrMkErr	-67	could not find a valid address mark
dataVerErr	-68	read verify compare failed
badCkSmErr	-69	address mark checksum was incorrect
badBtSlpErr	-70	one of the address mark bit slip nibbles was incorrect (GCR)
noDtaMkErr	-71	could not find a data mark header
badDCkSum	-72	bad data mark checksum
badDBtSlp	-73	one of the data mark bit slip nibbles was incorrect (GCR)
wrUnderRun	-74	could not write fast enough to keep up with the IWM
cantStepErr	-75	step handshake failed during seek
tk0BadErr	-76	track zero detect sensor does not change during a head recalibration
initIWMErr	-77	unable to initialize IWM
twoSideErr	-78	tried to read a double-sided disk on a single-sided drive
spdAdjErr	-79	unable to correctly adjust the drive speed (GCR, 400K drives only)
seekErr	-80	wrong track number read in a sector's address field
sectNFErr	-81	sector number never found on a track

Control Calls

Control calls perform all of the operations not related to reading from or writing to a particular disk associated with this driver. The control opcode is passed to the driver in the csCode field of the I/O parameter block (byte 26). Control calls which return information do so by passing it back, starting at the csParam field of the I/O parameter block (byte 28). Following is a description of each control operation with any result codes it returns.

Kill I/O (csCode=1)

Kill I/O is called to abort any current I/O request in progress. The Sony driver does not support this control call and always returns a result code of -1.

Verify Disk (csCode=5)

Verify Disk reads every sector from the selected disk to verify that they all have been written correctly. If any sector is found to be bad, it aborts immediately and returns one of the following error codes:

noErr	0	no error
controlErr	-17	verify failed (Hard Disk 20 only)
nsDrvErr	-56	no such drive number
noDriveErr	-64	drive not installed
noNybErr	-66	various read errors
badDBtSlp	-73	bad data bit slip error
cantStepErr	-75	step handshake failed during seek
initIWMErr	-77	unable to initialize IWM
pdAdjErr	-79	unable to correctly adjust disk speed
verErr	-84	track failed to verify

Format Disk (csCode=6)

If the selected disk is a floppy disk, Format Disk writes address headers and data fields for every sector on the disk (for GCR disks only) and does a limited verification of the format by checking that the address field of the first sector on each track can be read. If the selected disk is a Hard Disk 20, Format Disk does not do an actual format of the media, but instead initializes the data of each sector to all zeroes. If any error occurs (including write-protected media), Format Disk aborts the formatting and returns an error code.

The csParam field is used to specify the type of format to be done on floppy disks only. In pre-SWIM versions of the driver, putting a $0001 at csParam creates a single-sided disk, while a non-$0001 value (usually $0002) creates a double-sided disk. In the SWIM and later versions, this value is an index of a list of possible formats for the given hardware and disk combination (see the **Return Format List** (csCode = 6) status call for values).

noErr	0	no error
controlErr	-17	format failed (Hard Disk 20 only)
wPrErr	-44	disk is write-protected
paramErr	-50	format type is out of range
nsDrvErr	-56	no such drive number
noDriveErr	-64	drive not installed
noNybErr	-66	various read errors
badBtSlpErr	-70	…
wrUnderRun	-74	write underrun occurred
cantStepErr	-75	step handshake failed during seek
initIWMErr	-77	unable to initialize IWM
spdAdjErr	-79	unable to correctly adjust disk speed
fmt1Err	-82	cannot find sector zero after track format
fmt2Err	-83	cannot get enough sync between sectors
noIndexErr	-83	timed out waiting for drive's index pulse (MFM only)

Eject Disk (csCode=7)

Eject Disk ejects the disk in the selected drive if that drive supports removable media. Since Hard Disk 20 drives are not removable, if one is ejected, the driver posts a diskInserted event so that the operating system remounts the drive.

noErr	0	no error
nsDrvErr	-56	no such drive number
noDriveErr	-64	drive not installed
cantStepErr	-75	step handshake failed during seek
tk0BadErr	-76	track zero detect does not change during head recalibration
initIWMErr	-77	unable to initialize IWM

Set Tag Buffer (csCode=8)

If csParam is zero, then no separate tag buffer is used. If csParam is non-zero, it is assumed to contain a pointer to a buffer where tag bytes from each block are read into or written from on each prime call. Every time a block is read from the disk, the 12 tag bytes are copied into the file tags buffer at TagData+2 ($2FC) and then into the user's tag buffer. When a block is written, tag bytes are copied into the file tag buffer from the user's tag buffer, and then written to the disk with the rest of the block. The position of a particular block's tag bytes in the user tag buffer is determined by that block's position relative to the first block read or written on the **current** prime call. The file tags for GCR disks include information that a scavenging utility could use to rebuild a disk if the directory structure were trashed. Figure 1 illustrates the tags. For more detailed information about tag buffers, refer to *Inside Macintosh*, Volume II, The Disk Driver.

0	file number
4	fork type (bit 1=1 if resource fork)
5	file attributes (bit 0=1 if locked)
6	relative file block number
8	disk block number

Figure 1–File Tags

Track Cache Control (csCode=9)

When the track cache is enabled, all of the sectors on the last track accessed during a read request are read into a buffer in RAM. The sectors that were actually requested are also returned in the user's buffer. On future read requests, if the track is the same as the last read track, the sector data is read from the cache instead of the disk. Write requests to the driver are passed directly to the disk, and any of the sectors written that are in the cache are marked invalid. Two bytes are passed at `csParam` to control the cache:

csParam	csParam+1
=0: disable the cache	<0: remove the cache
≠0: enable the cache	=0: do not remove or install
	>0: install the cache

When the cache is removed, 680x0 register `D0` contains the previous size of the cache.

`noErr`	0	no error
`memFullErr`	-108	not enough room in heap zone to install track cache

Return Physical Drive Icon (csCode=21)

This call returns a pointer to an icon describing the selected drive's physical location. The supported drive icons are shown in Figure 2. Note that only the icons for a particular machine are included in that version of the driver. The Hard Disk 20 icon is in the drive's ROM, so it is available only when a Hard Disk 20 is connected.

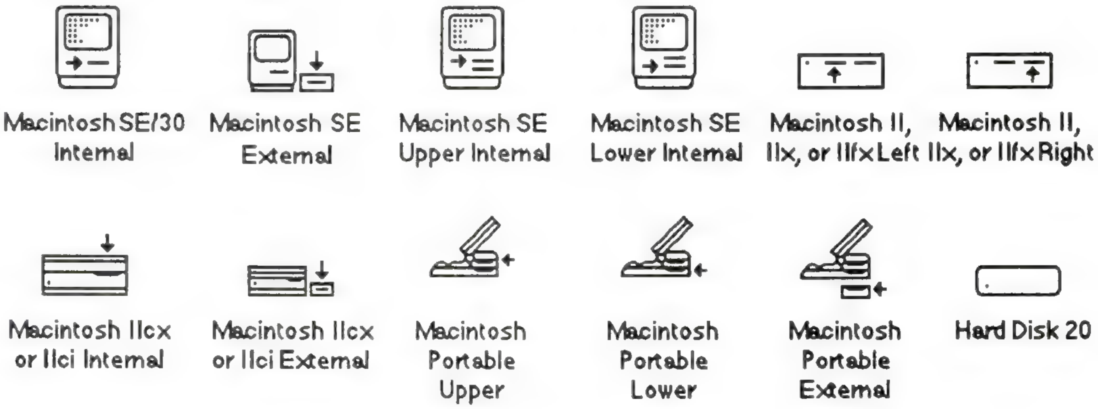

Macintosh SE/30 Internal
Macintosh SE External
Macintosh SE Upper Internal
Macintosh SE Lower Internal
Macintosh II, IIx, or IIfx Left
Macintosh II, IIx, or IIfx Right
Macintosh IIcx or IIci Internal
Macintosh IIcx or IIci External
Macintosh Portable Upper
Macintosh Portable Lower
Macintosh Portable External
Hard Disk 20

Figure 2–Physical Drive Icons

`noErr`	0	no error
`controlErr`	-17	icon does not exist or is not available (Hard Disk 20 only)
`nsDrvErr`	-56	no such drive number
`noDriveErr`	-64	drive not installed

Return Media Icon (csCode=22)

Return Media Icon returns a pointer to an icon for the selected drive's media type. The Sony floppy disk icon is stored in the driver, while the Hard Disk 20 icon is retrieved from the drive's ROM.

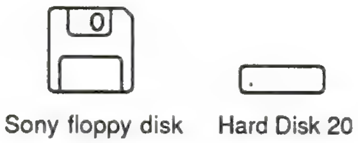


Figure 3–Media Icons

`noErr`	0	no error
`controlErr`	-17	icon does not exist or is not available (Hard Disk 20 only)
`nsDrvErr`	-56	no such drive number
`noDriveErr`	-64	drive not installed

Return Drive Info (csCode=23)

Return Drive Info returns a 32-bit value in `csParam` that describes the location and attributes of the selected drive.

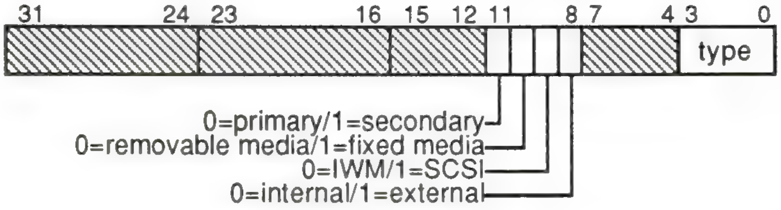


Figure 4–Return Drive Info in csParam

As illustrated in Figure 4, most of the bits of this returned value are currently not used and left open for future expansion. The drive type field occupies bits zero to three and describes the kind of drive that is connected. Currently six different "types" are supported:

0	no such drive
1	unspecified drive
2	400K Sony
3	800K Sony
4	SuperDrive (400K/800K GCR, 720K/1440K MFM)
5	**reserved**
6	**reserved**
7	Hard Disk 20
8-15	**reserved**

The attributes field occupies bits 8 to 11 and describes the location (internal or external, primary or secondary), drive interface (IWM or SCSI), and media type (fixed or removable).

`noErr`	0	no error
`nsDrvErr`	-56	no such drive number
`noDriveErr`	-64	drive not installed

Diagnostic Raw Track Dump (csCode=18244)

This control call reads all or part of a track and returns the raw data it finds so applications can access a floppy disk at a very low level without having to directly access the hardware. This call is available in the ROM of the Macintosh IIfx. An application should check for the presence of this call, and if it's not available, either bail out or find another way to read the raw data. If you make the call and it is not supported on the machine, then it returns with a -17 `controlErr`. This call is needed since the SWIM chip is **not** going to be directly addressable in the future. The following parameters are passed starting at `csParam`:

+0	`clockBitsBuffer`	longint	pointer to packed bit array (MFM disks only), or `NIL`
+4	`dataBuffer`	longint	pointer to raw track data, or `NIL`
+8	`byteCount`	longint	number of bytes requested (dataBuffer must be able to hold this many bytes)
+12	`numDone`	longint	number of bytes actually read (≤`byteCount`)
+16	`searchMode`	word	when to start collecting bytes: 0 = as soon as spindle motor is up to speed 1 = after reading an address field 2 = after reading a data field 3 = at the index mark (MFM disks only)
+18	`track`	word	which track to read (0-79)
+20	`side`	byte	which side to read (0-1)
+21	`sector`	byte	which sector to synchronize on (GCR=0-8,9,10,11; MFM=1-9 or 1-18); however, any value from 0-255 is okay

If `clockBitsBuffer` is not `NIL`, it points to a buffer that must be at least one-eighth the size of `dataBuffer`. It consists of an array of bits signifying whether or not the corresponding byte in `dataBuffer` is a mark or data byte. If a bit is equal to one, the byte is an MFM mark byte, but if it is equal to zero, the byte is an MFM data byte. The relationship between bits in `clockBitsBuffer` and `dataBuffer` is shown in Figure 5. The example shows a typical MFM address field.

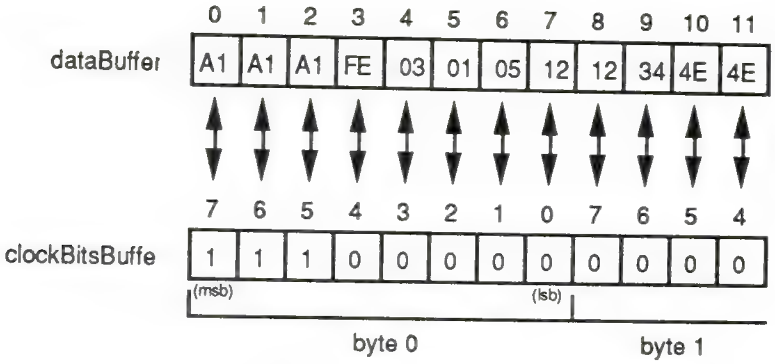


Figure 5–clockBitsBuffer versus dataBuffer

Note: If both `clockBitsBuffer` and `dataBuffer` are `NIL`, the call does nothing. This provides a way for applications to determine if the call exists without first having to allocate large buffers.

Note: The `clockBitsBuffer` has random data in it for GCR disks, but the `dataBuffer` has valid information.

`ByteCount` specifies the number of raw bytes to read. It may not be possible to read that many bytes on every Macintosh due to differences in the way that the hardware and software are implemented, so the call returns the number of bytes that were actually read in `numDone`. If `byteCount` is zero, the call does nothing.

`SearchMode` specifies when to begin actually collecting bytes. The first case (0) implies that the location where reading begins is somewhat random. Cases 1 and 2 begin reading bytes as soon after the end of an address or data field as possible. If the read is done on an MFM disk, the call resynchronizes and begins reading at the next mark byte that follows a sync field. The last case synchronizes with the drive's index signal and then begins reading as soon as it sees a mark byte that follows a sync field.

The `track`, `side`, and `sector` fields are self-explanatory. Of course, the sector number is not needed or used when `searchMode` is either 0 or 3.

noErr	0	no error
controlErr	-17	this call is not supported on the host Macintosh
paramErr	-50	one or more of the parameters is out of range
nsDrvErr	-56	no such drive number
noDriveErr	-64	drive not installed
offLinErr	-65	read or write request made to an ejected disk
noNybErr	-66	could not find five nibbles in 200 tries (GCR) or byte timeout (MFM)
noAdrMkErr	-67	could not find a valid address mark
badCkSmErr	-69	address mark checksum was incorrect
badBtSlpErr	-70	one of the address mark bit slip nibbles was incorrect (GCR)
noDtaMkErr	-71	could not find a data mark header
badDCkSum	-72	bad data mark checksum
badDBtSlp	-73	one of the data mark bit slip nibbles was incorrect (GCR)
cantStepErr	-75	step handshake failed during seek
twoSideErr	-78	tried to read a double-sided disk on a single-sided drive
spdAdjErr	-79	unable to correctly adjust the drive speed (GCR, 400K drives only)
seekErr	-80	wrong track number read in a sector's address field
sectNFErr	-81	sector number never found on a track
noIndexErr	-83	timed out waiting for index signal

Status Calls

The Sony driver currently supports three status calls, which are described in this section. As with the control calls, the status opcode is passed to the driver in the csCode field of the I/O parameter block (byte 26). The returned status information is passed back starting at the csParam field of the I/O parameter block (byte 28).

Return Format List (csCode=6)

Return Format List is only supported in SWIM or later versions of the Sony driver, whether or not MFM disks are supported. It returns a list of all possible disk formats that are supported with the current combination of disk controller, drive, and media. On entry, csParam contains a value specifying the maximum number of formats to return (refer to Technical Note #262, Controlling Status Calls, for more information), and csParam+2 contains a pointer to a table which contains the list. On exit, csParam contains the number of formats returned (no more than specified) and the table contains the list of formats. If no disk is inserted in the drive, the call returns a noDriveErr code. The format information is given in an eight-byte record as shown in Figure 6.

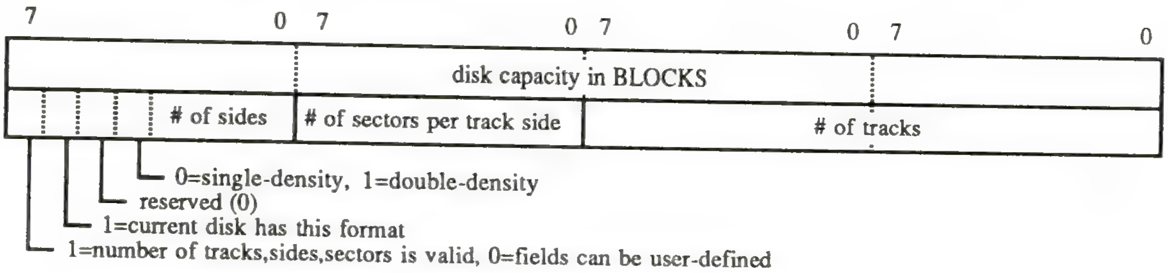

7
0 7
0 7
0 7
0
disk capacity in BLOCKS
# of sides
# of sectors per track side
# of tracks
0=single-density, 1=double-density
reserved (0)
1=current disk has this format
1=number of tracks,sides,sectors is valid, 0=fields can be user-defined

Figure 6–Format Information From Return Format List

If a `track`, `side`, or `sector` field is zero when the TSS valid bit is set to one, that field is considered to be a “don’t care” as far as describing the format of the disk. When the TSS valid bit is zero, the `track`, `side`, and `sector` fields may be driver-defined. The formats supported by the driver are as follows:

Format	Capacity in blocks	TSS valid	SD or DD	# of Sides	# of Sectors	# of Tracks
400K GCR	800	yes	SD	1	10[1]	80
800K GCR	1600	yes	SD	2	10[1]	80
720K MFM[2]	1440	yes	SD	2	9	80
1440K MFM[2,3]	2880	yes	DD	2	18	80
Hard Disk 20	38965	no	SD	0	0	0

[1] average number of sectors
[2] requires SWIM and SuperDrive
[3] requires HD media

`noErr`	0	no error
`paramErr`	-50	error in user's parameter list
`nsDrvErr`	-56	no such drive number
`noDriveErr`	-64	drive not installed

Drive Status (csCode=8)

Drive Status returns information about a particular drive starting at csParam. Drive Status returns only a noErr (0) message.

Offset	Name	Description
0	current track	
2	bit 7 set=write-protected	
3	disk-in-place?	<0 = disk is being ejected 0 = no disk is currently in the drive 1 = disk was just inserted but no read or write requests have been made for this disk 2 = OS has tried to mount the disk (i.e., read request to driver) 3 = same as 2, except that this is a high-density disk formatted as 400K/800K GCR 8 = same as 2 except for an Hard Disk 20 (8 means it's also non-ejectable)
4	drive installed?	–1 = no drive installed 0 = do not know 1 = drive installed
5	number of sides:	0 = single-sided; -1 = double-sided
6	drive queue element:	6 qLink: pointer to next queue element 10 qType: type of queue (drvQType) 12 dqDrive: drive number 14 dqRefNum: Sony driver's reference number 16 dqFSID: file system ID
18	two-sided format?:	0 = current disk has single-sided format -1 = current disk has double-sided format
19	new interface:	0 = old drive interface (400K) -1 = new interface (800K and later)
20	soft error count (2 bytes)	

Further Reference:

- *Inside Macintosh*, Volumes II, IV & V, The Disk Driver
- Technical Note #262, Controlling Status Calls.

#273: SCSI Termination

Revised by: Dave Radcliffe — May 1992
Written by: Rich "I See Colors" Collyer — April 1990

This Technical Note discusses SCSI termination on the Macintosh, including the new rules of termination which are necessary with the advent of the high-speed Macintosh IIfx.
Changes since June 1990: Added a discussion of Macintosh Quadra SCSI termination.

Why Is the Terminator After Sarah Connor?

One of the features of the Macintosh IIfx is a new SCSI chip that provides SCSI data transfer rates up to three megabytes per second, faster than any existing Macintosh model except the Macintosh Quadra. To achieve these transfer rates, components on the Macintosh IIfx logic board are smaller and faster, requiring different termination configurations than previous Macintosh models.

The Macintosh IIfx requires the use of a combination of the following three new termination parts. Users need to use these parts instead of existing SCSI termination parts to configure a IIfx with SCSI devices. The Macintosh Quadra does *not* require special termination as the IIfx does, but it does have some special rules of its own and these are discussed in a later section.

Apple SCSI Cable Terminator II. The Apple SCSI Cable Terminator II is a revised external terminator for the Macintosh IIfx. All finished goods Macintosh IIfx systems ship with this terminator in the box. It is easily recognized because of the black color. Under **no** circumstances should one use more than a single Apple SCSI Cable Terminator II on an external SCSI chain—doing so may **damage** the logic board.

Internal SCSI Termination Block. The Internal SCSI Termination Block provides internal termination resistance for Macintosh IIfx systems without internal hard drives. All finished goods systems shipping without internal hard drives have the Internal SCSI Terminator Block installed.

Internal SCSI Filter. The Internal SCSI Filter provides termination capacitance for internal Macintosh IIfx hard drives that shipped prior to March 19, 1990. All finished goods systems shipping without internal hard drives have the Internal SCSI filter installed.

The new termination configurations are simple, and you can remember them with a single rule: Macintosh IIfx systems with external SCSI chains require a terminator at both ends of the SCSI chain. One is internal to the system, while the second is external, located at the end of the chain.

The reason for the new terminator is that on the Macintosh IIfx and future hardware, the SCSI controller chip is a two micron part which makes it very fast. One of the results of this speed is that the chip now thinks that glitches in the /REQ line are real signals. This problem is not likely to occur on all of the Macintosh IIfx machines, but if you have a problem with your hard drive not getting mounted on the new machine, you should try a new terminator first. The symptom is more likely to show up on machines with several (three or more) external SCSI devices attached to the computer and long strands of SCSI cables. Figure 1 illustrates the old-style terminator with the signal showing the spike propagation.

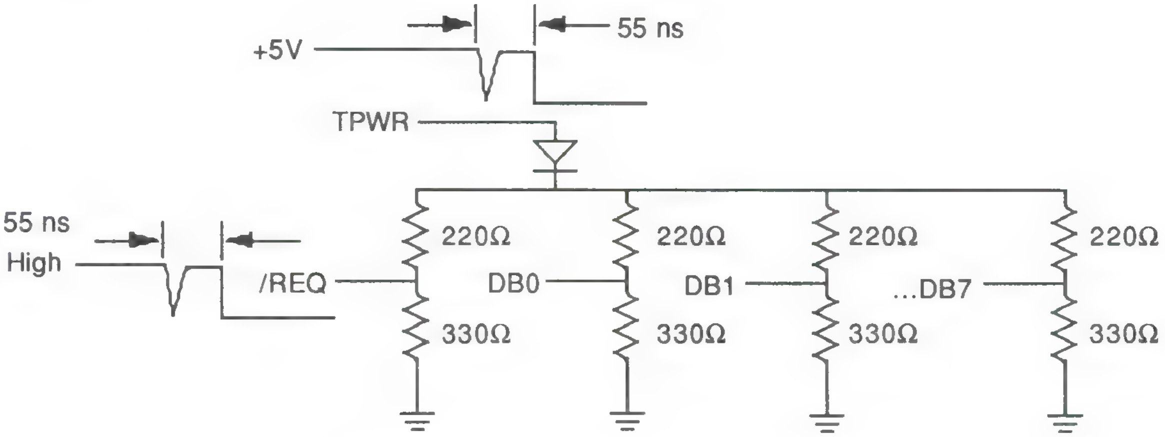


Figure 1 Old-Style Terminator (Gray)

Basically, if a majority of the data lines change state at once, there is a sudden drain on the TPWR line which is resistively coupled to all of the lines, including the /REQ line. This sudden drain causes a spike in the line, and this spike is propagated into the /REQ line and to the SCSI controller chip. The newer SCSI controller chip in the IIfx interprets this spike as a /REQ signal and starts reading data from the data lines; however, since the data lines need 55 ns to settle, the data that the controller chip reads is junk.

All internal hard disk drives sold by Apple with the IIfx and later machines have the Internal SCSI Filter installed; however, most third-party drives do not yet have this filter installed and must be modified by a qualified service provider to work correctly with the IIfx.

How to Stop the Terminator

Since the problem is caused by a drop in the TPWR line, the fix is to smooth out the line. One need only add a 2.2 µF capacitor and a 0.01 µF ceramic capacitor as illustrated in Figure 2. These capacitors act like a battery and provide a little extra current when it is needed. This extra current results in a smoother signal, which the SCSI controller chip does not interpret as a /REQ signal.

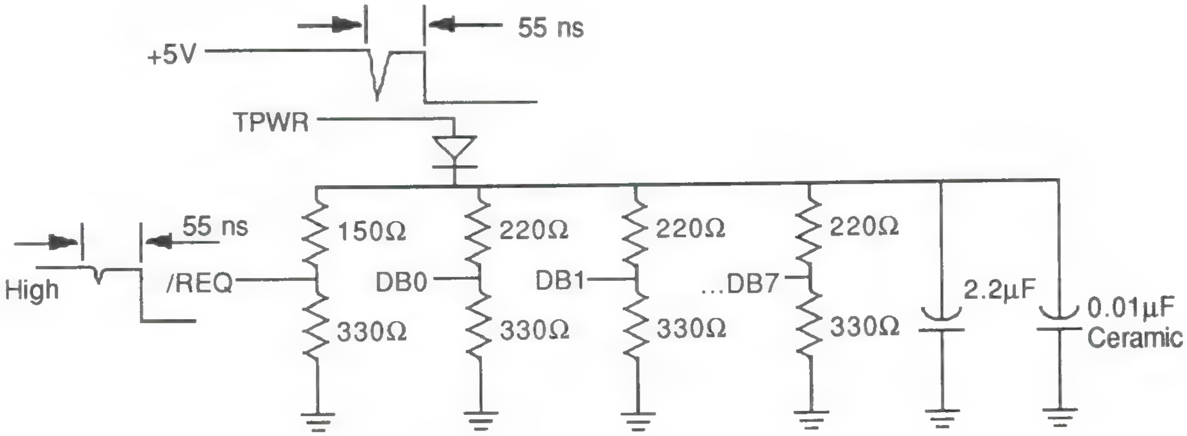


Figure 2 New-Style Terminator (Black)

This new type of filter is only for internal hard disk drives. The Macintosh IIfx ships with a new and improved external terminator (black in color), so hard drive manufacturers do not need to worry about external termination. Apple also ships an internal filter with every IIfx that handles the capacitance problem. This internal terminator has two parts. The first is the resistors for the terminator. This part should already be installed on all internal hard disk drives, so it is used only for CPUs that do not have an internal hard drive. The second part of the internal terminator is the capacitor filter. This filter should be installed on the hard disk drive end of the SCSI internal cable. If your hard drive implements the new capacitors, you can, and should, install the capacitor filter—you cannot have too much capacitance.

External Termination

If you manufacture an external SCSI device **do not** include termination in it. The only terminator that should be outside of a Macintosh IIfx is Apple's external terminator, and it should be at the end of the SCSI chain. If you make a SCSI terminator, it is most likely incompatible and may cause damage to the hardware or the data. If your SCSI device cannot connect with Apple's terminator, then you should provide an adapter that allows your SCSI device to attach to the provided terminator.

Note: A notice in the Macintosh IIfx finished goods box instructs customers to return self-terminating SCSI devices to the service provider to disable termination.

You're Terminated

Not every Macintosh IIfx owner is likely to experience this inconvenience, but a few will. If your customers report problems that appear to be termination related, then the first possible solution is to fix the terminator (for external devices) or implement the filter (for internal devices). If you manufacture an external SCSI device that is self-terminating, you should remove it. This incompatibility will continue with future hardware products and could even surface on the Macintosh IIci.

Macintosh Quadra Termination

Proper SCSI termination is critical for correct operation of the Macintosh Quadra computers, just as with all Macintosh computers. The Macintosh Quadra computers require external SCSI termination at the end of the device chain, either supplied by the last device in the chain, or using a standard Apple SCSI Cable Terminator (M0332LL/A). Note that this is the standard SCSI terminator, ***not*** the black terminator required by the Macintosh IIfx (although the black IIfx terminator may be used as well).

Termination is generally supplied at the factory for use with internal SCSI devices. Some early floppy-only Macintosh Quadra 700 units may not have internal termination, so users who attach external SCSI devices (without having added an internal SCSI device) may need to double terminate their external SCSI chain. Properly terminated floppy-only Macintosh Quadra 700 units will have a terminator inserted into the motherboard internal SCSI cable connector. Users of internal SCSI devices must, of course, remove this terminator before connecting their internal SCSI device.

The Macintosh Quadra 900 is the first Macintosh computer to provide a separate, internal SCSI bus. This bus is physically isolated from the external SCSI bus and must also be properly terminated. The cable provided with the machine includes all the termination necessary, so ***all*** internal devices must have SCSI termination removed before connecting to the internal Macintosh Quadra 900 SCSI cable. If extra termination is supplied it may cause intermittent hardware failures as well as physical damage to the device.

Developers who ship terminated SCSI devices for possible internal use in the Macintosh Quadra 900 must provide users with instructions for removing the termination.

Macintosh
Technical Notes

Developer Technical Support

#274: The Compleat Guide to TeachText

Written by: Bryan Stearns, DTS Emeritus April 1990

This Technical Note explains how to use TeachText to create release notes, complete with pictures, which every Macintosh owner can read. This Note assumes familiarity with ResEdit.

Background

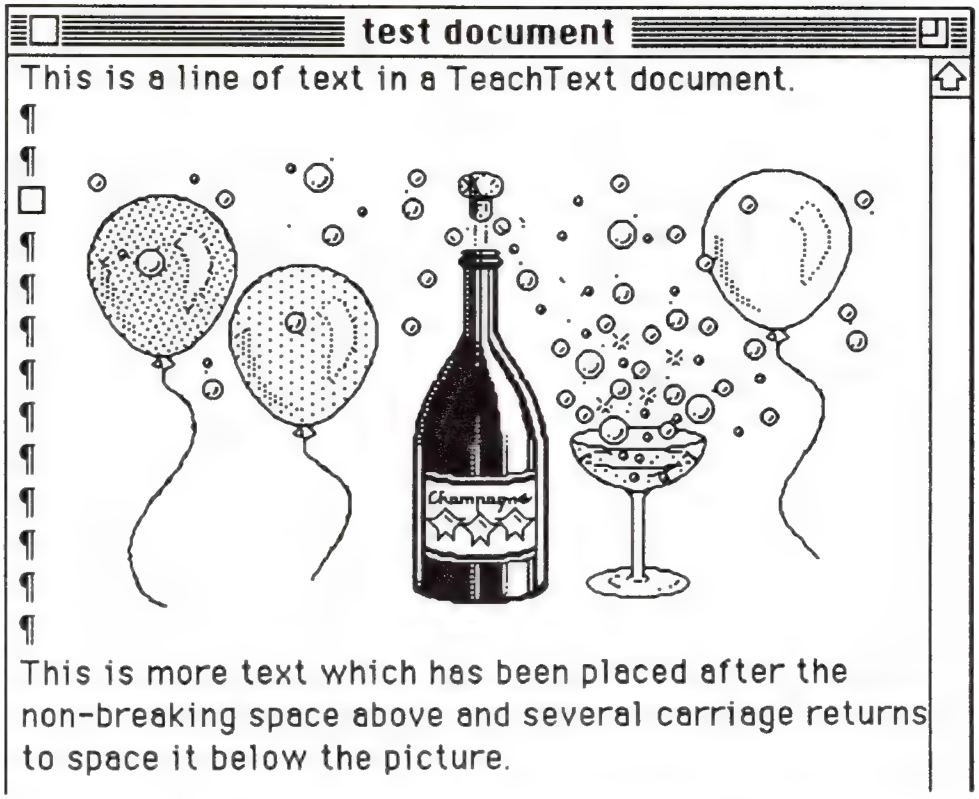
TeachText is two, two, two applications in one, and Apple ships it with every Macintosh. It's a simple text editing training tool with support for the standard editing primitives, saving and printing, and it's also a tool which allows every Macintosh owner to browse read-only release notes or other documents which may contain text and pictures.

Since TeachText only allows a single open document at a time, it uses the document's file type to determine which of the two applications it should be. If the file type is "TEXT" (as are all files created by TeachText), it operates as a simple text editor, but if the file type is "ttro" (lowercase is significant), it only allows the user to scroll through the document or print its contents—modifications are not allowed, thus making the file read-only.

How TeachText Handle Pictures

TeachText operates on documents of the two file types previously described, and either may contain pictures. However, pictures tend to disappear when editing the document in which they are contained (to those hardy souls attempting to create documents with pictures who must put up with this during the creation process, my apologies), thus all documents which contain pictures should be distributed as read-only (i.e., file type "ttro").

A document's pictures are stored as purgeable `'PICT'` resources in the resource fork of the document. Whenever a file is opened, each of these picture resources is loaded in numerical order, and its size is read into an array (so TeachText can later test to see if a picture needs to be drawn into the window without loading the picture). After the picture resources are loaded (and every time the window is resized thereafter), TeachText scans the text of the document for non-breaking space characters (ASCII $CA, entered as Option-Space Bar and usually used instead of a space to prevent related words from being split across line boundaries). In TeachText documents, a non-breaking space character represents the line on which the top of a picture resides. Figure 1 illustrates this relationship.



This is a line of text in a TeachText document.

¶

¶

□

¶

¶

¶

¶

¶

¶

¶

¶

¶

¶

This is more text which has been placed after the non-breaking space above and several carriage returns to space it below the picture.

Non-breaking space (invisible) places top of picture.

Carriage returns (invisible) place the text which follows after the picture.

Figure 1–Picture With Non-Breaking Space and Surrounding Text

If there are more non-breaking space characters than `'PICT'` resources, TeachText ignores the extra non-breaking spaces. Likewise, if there are more `'PICT'` resources than non-breaking space characters, TeachText ignores the extra `'PICT'` resources. Every time an update event occurs, TeachText checks each picture in the array, and if any of the pictures in the array overlap the current update region, it draws that picture.

As it happens, TextEdit is particularly messy about redrawing large portions of the screen when a user is entering text, and this makes editing documents with pictures rather clumsy. Since resizing the window causes another scan for non-breaking space characters as well as an update event, sizing the window in any way causes TeachText to “refresh” the pictures.

Creating Release Notes With TeachText

So how does one use TeachText to create release notes? It’s easy. Get those creative juices flowing, grab a cup of strong coffee (or your favorite highly-caffeinated beverage), and read on.

Write the Text

You can handle this part yourself. Use any word processor or text editor that supports saving to text-only files (i.e., those files of type “TEXT”). You can even use TeachText if you so desire. Don’t worry about fonts or styles, since TeachText only gives you the default application font in plain style. Don’t put carriage returns after each line either, since TeachText automatically wraps lines, just like a real word processor (the TeachText window conforms to the size of the current screen, so don’t depend on the breaks you see either). Don’t worry about non-breaking space characters at this point either; you’ll get a chance to add them later. Just think about what pictures you want (if you want them at all) and in what order you want them. When you are finished with the text, save a text-only file. If your word processor gives you the option of putting carriage returns after lines or after paragraphs, choose the after paragraphs option.

Draw the Pictures

First make a backup of your Scrapbook file (you should find it in your System Folder) if it contains anything you consider important. After backing it up, throw away the original copy (this makes things much easier later on in the process), but don't worry, if you made a backup you can use it to restore the original when finished. If you prefer, you can just rename the Scrapbook file, which effectively makes a backup copy.

Unfortunately, the ideal method for creating a picture involves both a paint program and a draw program. In addition, you should use Geneva 12 point font (or another System font like Monaco 9, Chicago 12, or Geneva 9) in your picture since that is the font that the rest of the text in the TeachText document uses. Once you are finished with your pictures, save them to a document, then do one of the following:

1. If you used a painting program to draw your pictures:

   Select your picture with a Lasso tool to ensure that only the minimum size of the image is copied. This takes up less space on disk and centers the picture in the document better for the user. Copy the picture then paste it into the Scrapbook. Repeat these steps for each individual picture you wish to include in the document.

2. If you used a draw program to draw your pictures:

   Copy each of your pictures into the Scrapbook. Launch a paint program, then copy each picture from the Scrapbook into the paint program. Once every picture is in a paint document, open the Scrapbook and clear each of your pictures from the Scrapbook. The Scrapbook should say "Empty Scrapbook" when you are finished (unless you did not start with a fresh Scrapbook). Now follow the procedure in the steps for a painting program to copy and paste each of your pictures back into the Scrapbook.

At this point, regardless of which program you originally used to create your pictures, they should all be in the Scrapbook and in bitmap form (after being copied with a Lasso tool from a paint program).

Because of a quirk in the Printing Manager and PostScript®, you have to perform one more step. Launch a draw program, then copy each picture from the Scrapbook into the draw program. Once every picture is in a draw document, open the Scrapbook and clear each of your pictures from the Scrapbook. The Scrapbook should say "Empty Scrapbook" when you are finished (unless you did not start with a fresh Scrapbook). Now copy each picture back to the Scrapbook. This process makes the pictures "transparent" when printed, and this is important to avoid a problem with white, horizontal stripes running through your pictures.

Add the Pictures

Now to add the pictures to the TeachText document. Launch ResEdit and open the text-only TeachText document (you may want to work on a backup copy). ResEdit may warn you that the file does not have a resource fork and opening it will create one. This is fine, since you want a resource fork. If ResEdit does not warn you, then the file already has a resource fork (this means that there may already be resources there).

If the ResEdit window you get (whose title is that of the document name) contains any four-letter words (no, not those four-letter words, but words like `'MPSR'`, `'ETAB'`, etc.) other than `'PICT'`, then you should select them and clear them from the document. If you have already added some pictures to this file (and are replacing some of them), you should be especially careful, since it is easy to accidently delete the wrong one.

Now open your Scrapbook file (the one with all the pictures in it). Its ResEdit window should contain `'PICT'`, `'SMAP'`, and `'vers'` resources. Select `'PICT'` (don't double-click), and copy this resource to the TeachText document by bringing it's window to the front and selecting Paste from the Edit menu. If you do this step correctly, your pictures and text should all be in the same document. Now save the TeachText document so you don't have to do this step again and close the Scrapbook.

Now you need to put the pictures into the proper numerical order so they show up in the correct order in the TeachText document. Numbering starts at 1000 (i.e., first picture should be 1000, second picture 1001, etc.). To order these pictures, double-click on the `'PICT'` in the TeachText document's window. You should get another window which contains each of the pictures you copied into this document. Use the scroll bar until you find the first picture you want to appear in the document. Select it (by clicking on it once), and choose the Get Info or Get Resource Info option to get information on the resource. ResEdit displays an information window about the selected resource with space to enter a name and an ID (there is already a random ID number assigned). Change the ID to 1000 and give the picture a name too (i.e., "Figure 1", etc.). Near the bottom of this window you can see the resource attributes. Be sure that the "Purgeable" attribute is checked, then close the window. Repeat this process for each succeeding picture, giving each a successive number (i.e., 1001, 1002, 1003, etc.). When you are finished with all of the pictures, save the file and quit ResEdit.

That is the difficult part; the rest is icing. Go get some more coffee or whatever it is you are drinking.

Edit the Text to Make It Look Pretty With the Pictures

Launch TeachText and open your document. Find the location where you want to place the first picture and position the text cursor there. Enter a carriage return or two (more if you want more space before the picture) then a non-breaking space character (Option-Space Bar, remember), which will be invisible. Now resize the window, and *voilà*, when the window redraws, your picture will be just below the non-breaking space character. Now enter as many carriage returns as necessary to provide space for the picture. When you enter the first carriage return, TeachText will erase the picture, so you will need to resize the window again to verify your spacing.

Once you have enough room for the first picture (you probably want to leave an extra blank line or two after it too), move on to the next desired picture location and repeat the process. Continue this process (and don't forget to save the document along the way) until you have placed all of the pictures. When you finish placing the pictures, you should save the document again and try printing it on both an ImageWriter and LaserWriter if possible. You may wish to tweak the picture spacing or location to keep them from crossing printed-page boundaries.

When you are satisfied with the results, Quit TeachText.

Make the File Read-Only

Make a copy of the file (to save a step if you decide to edit it again) then launch ResEdit. Now choose Get Info from the File menu and change the file type from "TEXT" to "ttro" (the lowercase is significant) and check to make sure the creator type is “ttxt”. Now quit ResEdit and save the changes to the document when prompted.

That’s all there is to it. (Now that wasn’t that bad, was it?)

A Few Hints On Creating Good Documents With Pictures

The following hints should help to make your TeachText document creation faster and more efficient as well as make the final document as nice as possible for the user.

- Always use the Lasso tool in paint programs to select pictures to appear in TeachText documents; it makes them smaller.

- Keep pictures as small and simple as possible; the document takes up less room on disk and scrolling is faster.

- If two pictures appear on top of each other, you probably have two non-breaking space characters on the same line. Simply delete one to fix it. It is generally a good idea to put non-breaking space characters on a line by themselves with a blank line before it. In addition, always leave room for an extra line after the picture so you do not have the picture running into the text which follows it.

- If you need to use the non-breaking space character as a non-breaking space, you can. Since TeachText assigns the numbered `'PICT'` resources to the non-breaking space characters in the document, you can simply skip a resource number to use a non-breaking space character as a non-breaking space in the text. For example, if you had four non-breaking spaces in the document and you wanted pictures at the first, second, and fourth, you would number your `'PICT'` resources 1000, 1001, and 1003. The third non-breaking space character would normally have `'PICT'` resource 1002 assigned to it, but since there is not a resource with this ID, it simply acts as a non-breaking space in the document.

- Do not worry about how horrible everything looks when you are editing; users will not be able to edit your document (unless they have read this Note), so they will not see the awful flashing, disappearing pictures, etc.

- Make the document read-only even if you do not use pictures. Distributing read-only documents to users gives the consistent impression that Release Notes are not to be modified.

- If your pictures are not appearing as you think they should, and if you cannot figure out what might be wrong by following the sequence in this Note, then try the following: Open the document with ResEdit. Click once on the `'PICT'` list and choose Open General from the File menu of ResEdit 1.x. You should get a window with a list of all of your pictures, in order, and numbered sequentially from 1000. If this is not what you get, then you have missed a step along the way and need to make sure all your pictures are in the resource and numbered sequentially.

Further Reference:

- *Macintosh ResEdit Reference*

PostScript is a registered trademark of Adobe Systems, Incorporated

#275: 32-Bit QuickDraw: Version 1.2 Features

Written by: Guillermo Ortiz

April 1990

This Technical Note describes the changes and enhancements to 32-Bit QuickDraw from version 1.0 (as shipped on the original Color Disk) to version 1.2, which ships with System Software 6.0.5 and later. This Note assumes familiarity with *Inside Macintosh*, Volume V, Color QuickDraw, and 32-Bit QuickDraw release notes.

32-Bit QuickDraw

Version 1.0 of 32-Bit QuickDraw shipped in May 1989 in response to the growing need for Color QuickDraw support for direct color devices and pictures (PICT2) and video boards for large-screen monitors which require 32-bit addressing for black and white operation. This original version of 32-Bit QuickDraw was a separate file that had to be copied manually into the System Folder. With the introduction of the Macintosh IIci, Apple put 32-Bit QuickDraw into ROM. Now System Software 6.0.5 and later offer 32-Bit QuickDraw as an integral part of the System Software which can be installed by the standard Installer (although the file is still separate).

This Note describes the changes and enhancements in version 1.2 of 32-Bit QuickDraw from version 1.0. Beginning with version 1.2, QuickDraw functionality is identical on all Color QuickDraw machines, including all the performance improvements which were originally only available in the IIci ROM.

New Features (In No Particular Order)

PICTs Contain Font Name Information

Every time you draw text inside of an `_OpenPicture` and `_ClosePicture` pair, QuickDraw stores the name of the current font and uses it when playing back the picture. The opcode used to save this information is $002C and its data is as follows:

```
PictFontInfo = Record
                 length   : Integer;    { length of data in bytes }
                 fontID   : Integer;    { ID in the source system }
                 fontName : Str255;
               END;
```

QuickDraw only saves this information one time for each font used in a picture. When QuickDraw plays back a picture, it uses the `fontID` as a reference into the list of font names which are used to set the correct font on the target system.

For example, the following code:

```
GetFNum('Venice', theFontID);     { Set a font before opening PICT}
TextFont(theFontID);

pHand2 := OpenPicture (pictRect);
    MoveTo(20,20);
    DrawString(' Better be Venice');

    GetFNum('Geneva', theFontID);
    TextFont(theFontID);
    MoveTo(20,40);
    DrawString('Geneva');

    GetFNum('New York', theFontID);
    TextFont(theFontID);
    MoveTo(20,60);
    DrawString('New York');

    GetFNum('Geneva', theFontID);
    TextFont(theFontID);
    MoveTo(20,80);
    DrawString('Geneva');
ClosePicture;
```

generates a picture containing font information like this:

```
OpCode 0x002C {9,
    "0005 0656 656E 6963 65"}           /* save current font     */
TxFont 'venice'
DHDVText {20, 20, " Better be Venice"}
OpCode 0x002C {9,                       /* save next font name   */
    "0003 0647 656E 6576 61"}
TxFont 'geneva'
DVText {20, "Geneva"}
OpCode 0x002C {11,                      /* ditto                 */
    "0002 084E 6577 2059 6F72 6B"}
TxFont 'newYork'
DVText {20, "New York"}
TxFont 'geneva'                         /* second Geneva does not
                                          need another $002C guy */

DVText {20, "Geneva"}
```

This feature works regardless of the type of picture being saved, including old style PICTs in a black and white port. Using _OpenCPicture instead of _OpenPicture to start a recording session results in the same functionality.

Direct PixPat Structures Now Supported

QuickDraw now supports 16-bit and 32-bit per pixel PixPat structures (patType = 1). In addition, it now supports a new patType (3) which uses dithering whenever 16-bit or 32-bit pixel patterns are displayed on indexed devices.

Direct 'cicn' Resources Now Supported

QuickDraw now supports 16-bit and 32-bit per pixel 'cicn' resources. The 16-bit per pixel is particularly cool since you save the space required for an 8-bit 'clut'.

GWorlds Can Now Be Allocated in MultiFinder Temporary Memory

You can now use the new `useMFTempBit` (bit 2) in a call to `NewGWorld` as an option to allocate pixels in MultiFinder temporary memory. In addition, you can now allocate screen buffers in MultiFinder temporary memory using the following routine, defined in Pascal and C:

```
FUNCTION NewTempScreenBuffer (globalRect: Rect; purgeable: BOOLEAN;
                              VAR gdh: GDHandle;
                               VAR offscreenPixMap: PixMapHandle): QDErr;
             INLINE $203C,$000E, $0015,$AB1D; { Move.L #$000E0015,D0
                                                 _QDOffscreen
                                               }

pascal QDErr NewTempScreenBuffer (Rect *globalRect, BOOLEAN purgeable,
                              GDHandle *gdh,
                              PixMapHandle *offscreenPixMap)
             ={0x203C,0x000E,0x0015,0xAB1D};
```

Indexed to Indexed Dithering

`_CopyBits` now supports the `ditherCopy` transfer mode whenever the destination device is between one and eight bits per pixel, regardless of the depth of the source image. With this support, an eight-bit image can now be approximated on a one-bit or a four-bit device by using error diffusion. Furthermore, an eight-bit image could also be dithered to a different set of 256 colors or a four-bit image could be dithered to an eight-bit device that does not have the desired colors.

32-Bit Addressed PixMap Structures

Version 1.2 defines a new `pmVersion` (`baseAddr32 = 4`) for 32-bit pointer `baseAddr` values. The `baseAddr` of such `PixMap` structures is treated as a 32-bit address, so no stripping or address translation is performed on it in 32-bit mode. This is a specially useful feature when the base address of a `PixMap` points to a NuBus™ address, for example in a video grabber board.

A new call, `Pixmap32Bit`, is now available to inquire if a given `PixMap` requires 32-bit addressing.

```
FUNCTION Pixmap32Bit(pmh:pixMapHandle):Boolean;
    INLINE $203C,$0004, $0016,$AB1D; { Move.L #$00040016,D0
                                        _QDOffscreen
                                      }

pascal BOOLEAN Pixmap32Bit(pixMapHandle pmh)
    = {0x203C,0x0004, 0x0016,0xAB1D};
```

Updated GetPixBaseAddress

Version 1.2 updates `GetPixBaseAddress` to return the address of any `PixMap`. The routine does the right address translation or stripping for all `PixMap` structures, including screen devices, unlocked GWorlds, and 32-bit addressed `PixMap` structures. The address it returns is only valid in 32-bit addressing mode. Also unless the `PixMap` is locked and made unpurgeable, the address returned by `GetPixBaseAddress` is only valid until any call to QuickDraw or the toolbox is made.

_CopyBits from Screen Devices

The picture recording mechanism has changed so that if you call `_CopyBits` while recording a picture with the source `PixMap` being a screen device, the data is correctly accumulated into the picture. Note that if the screen being copied is not the main screen, then the `PixMap` must be a 32-bit addressed `PixMap`. No auxiliary screen buffer is allocated if the source rectangle covers only one screen.

New Picture Recording Trap

Version 1.2 adds a new call, `_OpenCPicture`, to create pictures that contain information regarding the native resolution of the recorded image. When QuickDraw draws this picture, it scales the image to the resolution of the target device. Applications that need to scale the images directly can also access this information.

```
FUNCTION OpenCPicture(VAR CPictInfo:CPictRecord):PicHandle;
    INLINE $AA20;

pascal PicHandle OpenCPicture(CPictRecord *CPictInfo)
    =  0xAA20;
```

where

```
struct CPictRecord {
      Rect CPicFrame;          /* Bounding rect of Picture at native resolution */
      Fixed CPicHRes;          /* native horizontal resolution in pixels/inch   */
      Fixed CPicVRes;          /* native vertical resolution in pixels/inch     */
      short CPicVersion;       /* version of this PICT info set to -2           */
      short reserved;          /* for future expansion set to zero              */
      long reserved;           /* for future expansion set to zero              */
        };
```

The new picture header data looks like the following:

Size in bytes	Name	Description
2	`picSize`	low word of picture size
8	`picFrame`	bounding box at 72 dpi
Picture Header		
2	`version op`	version opcode = $0011
2	`version`	version number = $02FF
2	`Header op`	header opcode = $0C00
2	`version`	-2 for PICTs created with `_OpenCPicture`
2	`reserved`	
4	`HRes`	native horizontal resolution (`Fixed`)
4	`VRes`	native vertical resolution (`Fixed`)
8	`SrcRect`	native source rectangle
4	`reserved`	

The following is a sample PICT created with _OpenCPicture:

```
00 48                          /* low word of size                */
00 00 00 00 00 7D 00 7D        /* picFrame at 72 dpi              */
00 11                          /* PICT version opcode             */
02 FF                          /* version number                  */
0C 00                          /* PICT header Opcode              */
FF FE                          /* PICT version -2                 */
00 00                          /* reserved                        */
01 20 00 00                    /* HRes (Fixed)                    */
01 20 00 00                    /* VRes (Fixed)                    */
00 00 00 00 01 F4 01 F4        /* picFrame at native resolution */
00 00 00 00                    /* reserved                        */
/* picture data follows                                           */
00 FF                          /* end of picture opcode           */
```

Random Notes

For information on bug fixes in the System Software 6.0.5 release of 32-Bit QuickDraw (version 1.2), please refer to the System Software 6.0.5 Change History, which is available on the Developer CD Series, AppleLink in the Developer Services Bulletin Board (Developer Services: Macintosh Developer Technical Support: System Software), and the Apple FTP site on the Internet in the ~ftp/pub/dts/sw.license.

Note that the dispatching mechanism for the new _QDOffscreen calls is slightly different than previously documented; it now requires that the high word passed in D0 contain the total length of the parameters (in bytes). The reason for this change is that if the call is made in an earlier version of 32-Bit QuickDraw, the system can strip the parameters from the stack and return QDError set to the caller (instead of crashing).

Further Reference:

- *Inside Macintosh*, Volume V, Color QuickDraw
- 32-Bit QuickDraw Release Notes (available from APDA)
- System Software 6.0.5 Change History
- *develop*, Issue I

NuBus is a trademark of Texas Instruments.

#276: Gimmie Depth Or Gimmie Death (So You Want to be a Monitors Impersonator?)

Revised by: Guillermo Ortiz June 1990
Written by: Guillermo Ortiz April 1990

This Technical Note describes two new system calls that allow an application to change the depth and flags for a given device and also check whether a device supports a particular depth and flags setting. Apple provides these calls to give developers a better way to help **users** make changes when they consider it appropriate. Abusive use of these calls is a sure way to guarantee that the Thought Police come after you to confiscate your Macintoshes, your stock of Mountain Dew®, and your Technical Notes binder. This Note assumes familiarity with *Inside Macintosh*, Volume V, Graphics Devices.

Changes since April 1990: Corrected trap addresses and dispatch numbers in the `SetDepth` and `GetDepth` definitions.

Historic Novella

Since the introduction of the Macintosh II, developers have had the strong urge to change the depth of Macintosh screens under program control. Developers often ask, “How can I change the depth from my application like the Monitors cdev does?” The reasons for this question have varied from pure Macintosh hacking spirit to valid reasons to the lack of finding a good solution which would work regardless of the depth setting the user may choose.

A poor scenario occurs when a developer wants to impose a certain depth and color or black and white setting because the application does not work well, if at all, with any other configuration. The responses from DTS always include questions about what happens to the application when the system in use does not support this “optimal” configuration or when one monitor is set to the magic configuration, but others are not, or when the user brings the application to the front and it finds that the user has changed the setting to something with which it is not equipped to deal.

On the other hand, DTS does see situations where an application that deals with certain image types may work better with a particular setting and would like to present the user with a dialog box similar to the Monitors cdev to allow the user to change the depth and color settings from within the application.

Not everyone agrees on the wisdom of providing facilities for an application to allow users to change depth and color settings from within itself, but all agree that a well-behaved application (remembering that well-behaved applications are more likely to survive system and hardware changes) should only change depth and color settings in the following circumstances:

- Depth changes can **only** be made with the user's consent; never change depths because it is simply convenient for the application. The user paid for his system and if the user wants color, be prepared to give color. If the user wants millions of colors, don't change the display to one-bit black and white.

- The minimal amount of user input to change depth and color settings should be to provide a preferences dialog box where the user would be given a yes or no choice to change depths when a particular action is chosen. The application should make the "no" choice the default and have a sensible mechanism for handling the situation when the user chooses no.

- Under **no** condition should an application change depth or color settings while in the background. An application should only initiate depth or color changes when it is the frontmost application; do not twiddle with the user's settings while in the background.

The Calls

Beginning with System Software 6.0.5 (regardless of whether or not 32-Bit QuickDraw is installed), applications can make a call to `SetDepth` to change the depth and flag settings for a given device.

```
FUNCTION SetDepth(gd:GDHandle; newDepth,whichFlags, newFlags: Integer):Integer;
    INLINE $203C,$000A, $0013,$AAA2; { Move.L #$000A0013,D0
                                       _PMgrDispatch
                                     }

pascal short SetDepth(GDHandle gd, short  newDepth,short whichFlags, short newFlags)
                     = {0x203C, 0x000A, 0x0013, 0xAAA2};
```

Where `gd` is the device to be changed, `newDepth` is the desired depth (you can pass the bit depth or the mode necessary to set the requested depth,) `whichFlags` is a bit field selector specifying which bits in `newFlags` are meaningful, and `newFlags` are bits to be set in the `gdFlags` field of the device record as specified by `whichFlags`. For example, if you want to set a depth of eight in black and white, the call would be as follows:

```
someResult := SetDepth(myGDevice,8,1,0);
```

In this call, `newDepth = 8` sets an eight-bit depth, `whichFlags = 1` indicates that only bit one of `newFlags` is important, and `newFlags = 0` clears the `gdDevType` flag in the device record (0 = monochrome, 1 = color). `SetDepth` returns zero if successful or a non-zero value if it cannot impose the desired depth on the requested device.

Also beginning with System Software 6.0.5, applications can make a call to `HasDepth` to verify if a given device supports a mode for the desired depth.

```
FUNCTION HasDepth(gd:GDHandle; newDepth,whichFlags, newFlags: Integer):Integer;
    INLINE $203C,$000A $0014,$AAA2; { Move.L #$000A0014,D0
                                      _PMgrDispatch
                                    }

pascal short HasDepth(GDHandle gd, short  newDepth,short whichFlags, short newFlags)
                       = {0x203C,0x000A 0x0014,0xAAA2};
```

Where `gd` is the device to be verified, `newDepth` is the desired depth, `whichFlags` is a bit field selector specifying which bits in `newFlags` are meaningful, and `newFlags` are bits to be checked in the `gdFlags` field of the device record as specified by `whichFlags`. `HasDepth` returns zero if the desired depth or flag setting is not supported on the given device; otherwise, `HasDepth` returns the `mode` necessary to set the device to the desired depth (which may be passed as the `newDepth` parameter in a call to `SetDepth`).

Further Reference:

- *Inside Macintosh*, Volume V, Graphics Devices
- *Designing Cards and Drivers for the Macintosh Family*, Second Edition

Mountain Dew is a registered trademark of Pepsico, Inc.

#277: Of Time and Space and _CopyBits

Written by: Forrest Tanaka

June 1990

This Technical Note describes the various factors that can influence the speed of `_CopyBits` so that developers can set up conditions to achieve the best performance for the particular situation.

Can You Influence the Speed of _CopyBits?

`_CopyBits` has never been an "easy" QuickDraw routine, like `_LineTo` or even `_OpenPort`. Most programmers who are just beginning to adjust themselves to the Macintosh usually have to give `_CopyBits` a few tries before the right bits copy to the right places. Even many who feel that they have *become* Macintosh programmers still see reflections in their monitors of furrows between their eyebrows as they begin to press the key labelled "C."

`_CopyBits` is one of those routines that is so full of subtlety, it has the beginnings of something that could be considered to be personality. One subtlety involves the second most important thought that's on the minds of any computer programmer: execution speed. Why is `_CopyBits` fast? Why is it slow? Can I influence its speed? Is there really a clandestine state of reason? Is there a price to speed?

Influences on the Speed of _CopyBits

Yes, you can influence the speed of `_CopyBits`. Yes, it's even predictable. And yes, it's possible that you have to compromise to get the maximum speed. This Note is intended to give you a deeper understanding of the ways that the speed of `_CopyBits` can be affected; and hopefully you can then set up conditions for a `_CopyBits` call without the disturbing notion that someone else might be doing the same thing just a little bit better than you.

This Note talks about every factor that affects the speed of `_CopyBits` that I can think of and that can be reasonably controlled by a programmer or the person using an application. There are other factors not mentioned in this Note because I felt that they were just too esoteric to describe with any meaning.

In each case, this Note tries to give real-life examples showing the effect of each factor. These examples are just to give you a relative idea of the importance of each effect. In real life, the effects of the different factors give results that could be a lot different from the results presented in this Note. Each example is based on 100 `_CopyBits` calls from an off-screen pixel map to the screen on a Macintosh IIcx with an Apple Extended Video Card which is running System Software 6.0.5 and 32-Bit QuickDraw 1.2. The off-screen pixel map is eight bits deep with the standard eight-bit color table and 256 pixels high by 256 pixels wide. The screen is also in eight-bit color mode. Calling `_CopyBits` to copy the entire off-screen pixel map to the screen 100 times takes 204 ticks, and this Note refers to this figure as the "standard test." Since a tick on a Macintosh is approximately 1/60 of a second, the standard test runs at slightly less than 30 frames per second. As this Note discusses each factor, it presents an example with that factor changing and all other factors remaining the same as the standard test, which allows you to compare performance of the changed factor to that of the standard test of 204 ticks.

What follows is a discussion of each factor that can influence the speed of `_CopyBits`, in no particular order.

Dimensions of the Copied Area

One of the most obvious factors has to do with the dimensions of the copied area. `_CopyBits` takes as parameters two rectangles which specify the portion of the source pixel map from which you want to copy and the portion of the destination pixel map to which you want to copy it. All other factors being equal, the larger the rectangles, the more pixels `_CopyBits` has to copy and the longer it takes to do the job. To keep `_CopyBits` as fast as possible, copy the smallest rectangle possible.

Modifying the standard test so that `_CopyBits` only copies a 128-pixel wide by 128-pixel tall area produces a result of 109 ticks, which compares to the 204 tick performance for a 256-pixel wide by 256-pixel tall area.

QuickDraw is usually faster drawing wide things than it is drawing tall things, because consecutive pixels in memory are displayed horizontally. Drawing a series of pixels that are next to each other horizontally is easy because QuickDraw simply has to set consecutive memory locations, while drawing a series of pixels that are next to each other vertically is just a little bit harder because the address of each pixel must be calculated. `_CopyBits` is no exception to this general rule; it copies a row of pixels, goes to the next row, copies that row, goes to the next row, and so on. The time spent going between rows is a lot more than the time going between pixels on one row, so the effect is that `_CopyBits` is faster copying a short and wide section of a pixel map than it is copying a tall and narrow one. To keep `_CopyBits` as fast as possible, copy the shortest rectangle possible.

Modifying the standard test again so that the source and destination rectangles are 256 pixels wide by 50 pixels tall produces a result of 110 ticks, while modifying it so that the source and destination rectangles are 50 pixels wide by 256 pixels tall results in a time of 123 ticks. These 13 ticks may not seem like a big deal, but combined with other factors, there may be a case where they make a big difference.

Shape and Size of the Clip, Visible, and Mask Regions

`_CopyBits` always makes sure that it stays within the lines, so to speak. `_CopyBits` copies pixels clipped to the `maskRgn` that you pass as the last parameter to the call. If the destination is the current `GrafPort`, `_CopyBits` additionally clips to a region that's the intersection of the `clipRgn` and `visRgn` of the port. If the intersection of these three regions is not rectangular, then `_CopyBits` has to check each pixel to make sure it falls within the intersection, and this check slows `_CopyBits` down. If the intersection of these three regions is rectangular, then `_CopyBits` takes the fast case of copying constant-sized rows. To keep `_CopyBits` as fast as possible, make sure the intersection of the `clipRgn` and `visRgn` of the destination `GrafPort` and the `maskRgn` is rectangular. Of course, if the destination `GrafPort` is a window, then the `visRgn` is under the user's control.

In general, if the region that you are copying into has straight vertical edges for the most part, the time penalty of using a non-rectangular region is not that bad. Regions that only have small portions that are straight and vertical are the ones that slow `_CopyBits` down in a big way. Regions that are twisted or that have holes or islands can also have a big effect upon the speed, depending upon how complicated they are. As a rule of thumb, if a region looks like it slows `_CopyBits`, it probably does.

Modifying the standard test so the `maskRgn` is set to a circle that inscribes the example pixel map results in a time of 303 ticks, which is considerably longer than the standard test result of 204 ticks that involved copying a much larger area. Modifying the `maskRgn` to a square with 226 pixels per side, which has about the same total area of the circle just used, results in a time of 176 ticks.

Transfer Modes

Macintoshes without Color QuickDraw have eight transfer modes that work with `_CopyBits`, while those Macintoshes with Color QuickDraw get an additional nine modes. Because the algorithms for each of these modes can be pretty different from the others, the time it takes `_CopyBits` to work with each of these modes can vary radically. For several of these modes, the speed of `_CopyBits` can vary a lot depending upon the particular image being copied and the image over which this image is copied. It can also vary non-linearly depending upon the depth of the pixel maps. The arithmetic modes in particular are highly optimized for 32-bit deep pixel maps.

The standard test copies a fairly average-looking ray-traced image to a white background. Modifying the standard test to erase the background between each of the 100 calls to `_CopyBits` produced the following results for the modes listed (the tests were obviously also changed to reflect the proper mode. In addition, to make the results a little more meaningful, the time it took to erase the background has been subtracted from each result.

`srcCopy`	204	`notSrcCopy`	469	`addOver`	1500	`adMax`	1504
`srcOr`	436	`notSrcOr`	444	`addPin`	1514	`adMin`	1501
`srcBic`	441	`notSrcBic`	441	`subOver`	1493	`blend`	1553
`srcXor`	438	`notSrcXor`	436	`subPin`	1525	`transparent`	1107
						`hilite`	3127

Of course, the amount of time taken by some of these modes can be changed by changing the image to copy and the image over which it is copied. These figures are just to give an idea of how fast or slow some of these modes are in this particular situation.

There is actually one more mode which is not mentioned: `ditherCopy`. Apple introduced this mode with 32-Bit QuickDraw, and it makes `_CopyBits` do error-diffusion dithering when copying a pixel map from one depth to a pixel map of a lesser depth or to a pixel map of the same depth with a different color table. The speed of this transfer mode can be very fast or very slow, depending upon what pixel depths and colors are used and the particular image being copied. The `ditherCopy` mode is not included in the table since the range of figures is potentially very large; play with it and see for yourself. For more information about this mode, refer to the Color QuickDraw chapter in *Inside Macintosh*, Volume VI and the 32-Bit QuickDraw Developers' Notes from APDA.

Colorization

There is a variation of _CopyBits if the destination pixel map is the current port and the foreground color is not black or the background color is not white. If this is the case, then the source image is colorized when it's copied. For details, see Technical Note #163, Adding Color with _CopyBits. Because this colorization requires extra processing, _CopyBits slows down. To keep _CopyBits as fast as possible, make sure the foreground color is black, the background color is white, and that the current GDevice pixel map's color table has white in the first position and black in the last position.

Modifying the standard test so that the foreground color is pure red and the background color pure blue produces a result of 579 ticks.

Pixel Alignment

The alignment of pixels in the source pixel map relative to their alignment the destination pixel map can be surprisingly important to the speed of _CopyBits, but what is pixel alignment? Following is an example to demonstrate the concept of pixel alignment. Imagine you want to perform a _CopyBits on a one-bit-per-pixel off-screen pixel map into a window on a one-bit-per-pixel screen, and the window is three pixels from the left edge of the screen.

If you copy the entire off-screen pixel map to the left edge of the window, then _CopyBits must realign the pixels. Since the leftmost pixels of the off-screen pixel map are on a byte boundary, but the left edge of the window is three pixels away from a byte boundary, _CopyBits has to shift (or realign) each byte from the off-screen pixel map by three pixels before placing it on the screen. The process of aligning the pixels slows down _CopyBits.

Figure 1 shows an example of this realignment. An off-screen bit map specified by a pointer to a BitMap called offScreen is being copied to a window specified by a WindowPtr called window. window, which is 256 pixels wide and 256 pixels high, is positioned 50 pixels from the top of the screen and three pixels from the left edge of the screen. The screen has 512 pixels horizontally and 342 pixels vertically. The source rectangle that is passed to _CopyBits is sourceRect and the destination rectangle is destinationRect. Because offScreen is misaligned by three pixels, _CopyBits has to shift offScreen by three pixels before placing the image on the screen.

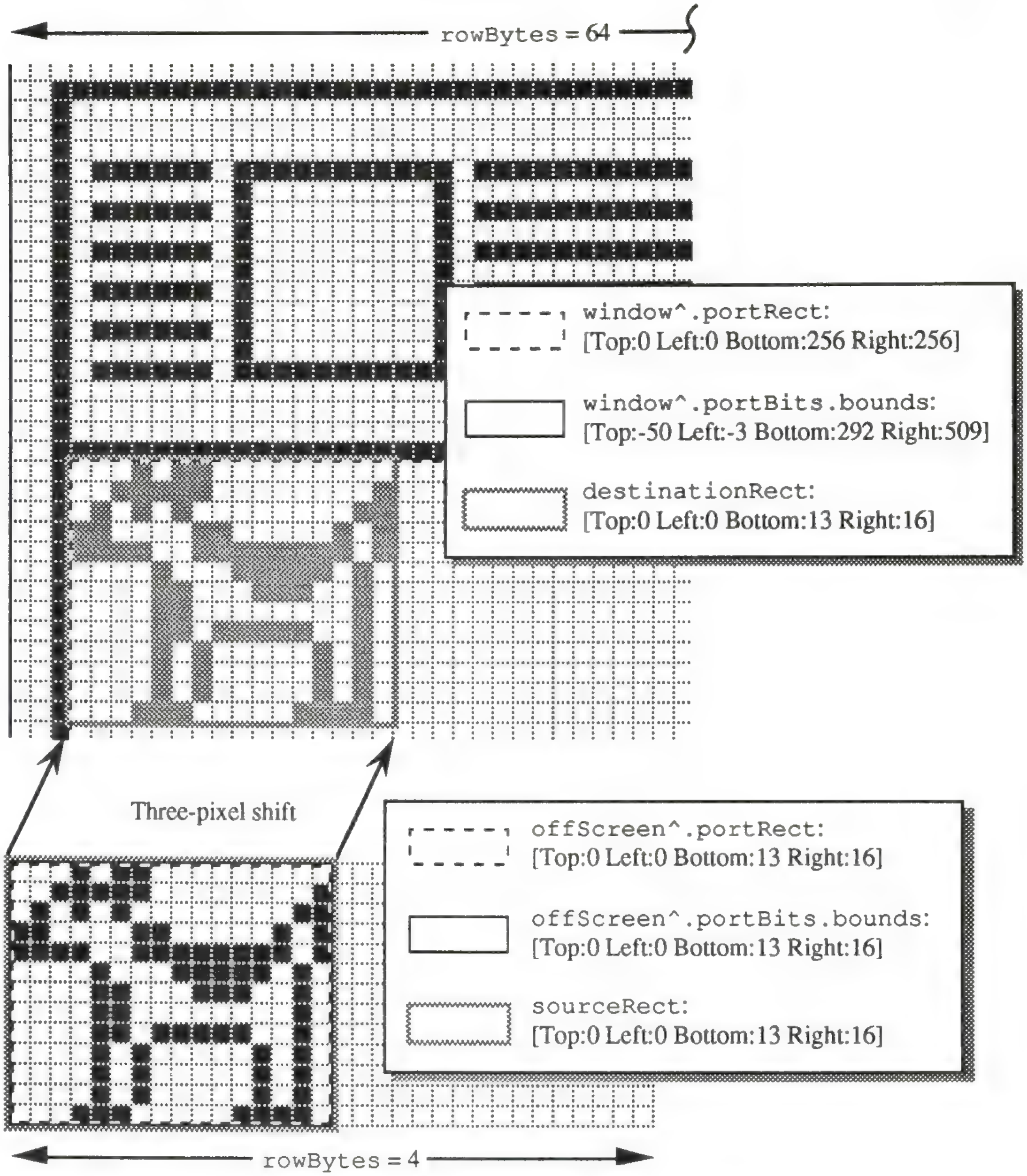

rowBytes = 64
window^.portRect:
[Top:0 Left:0 Bottom:256 Right:256]
window^.portBits.bounds:
[Top:-50 Left:-3 Bottom:292 Right:509]
destinationRect:
[Top:0 Left:0 Bottom:13 Right:16]
Three-pixel shift
offScreen^.portRect:
[Top:0 Left:0 Bottom:13 Right:16]
offScreen^.portBits.bounds:
[Top:0 Left:0 Bottom:13 Right:16]
sourceRect:
[Top:0 Left:0 Bottom:13 Right:16]
rowBytes = 4

Figure 1–offscreen Needs Realignment

By adjusting the off-screen pixel map so that its leftmost pixels are also three pixels away from a byte boundary, `_CopyBits` can just copy the bytes without shifting, which is a lot faster. This example holds true on all Macintosh models, whether they have Color QuickDraw or not. To keep `_CopyBits` as fast as possible, make sure the pixels in memory are aligned with the pixels on the screen. Figure 2 shows the same situation as Figure 1, except that `offScreen` is now properly aligned to `window`.

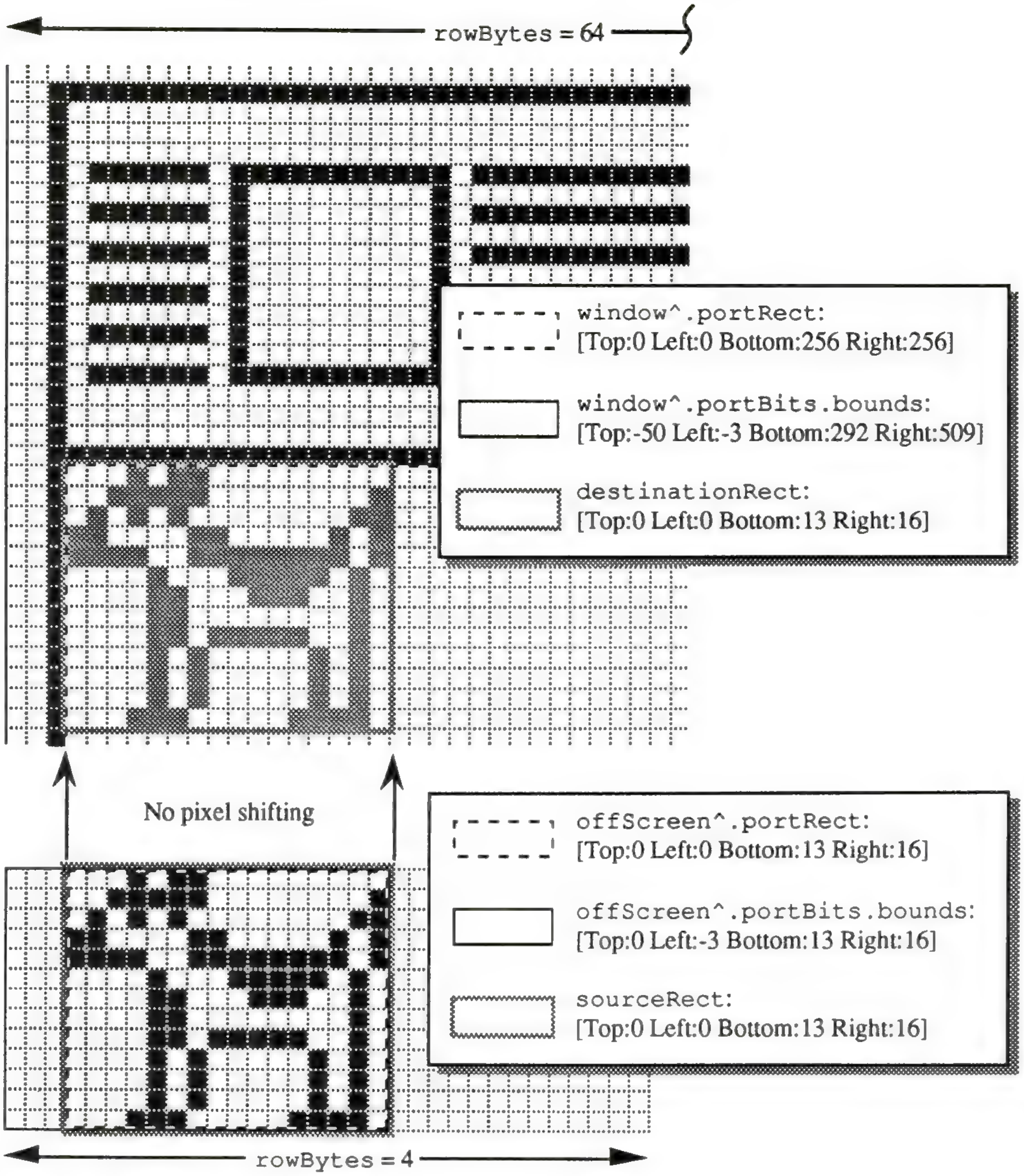

rowBytes = 64
window^.portRect:
[Top:0 Left:0 Bottom:256 Right:256]
window^.portBits.bounds:
[Top:-50 Left:-3 Bottom:292 Right:509]
destinationRect:
[Top:0 Left:0 Bottom:13 Right:16]
No pixel shifting
offScreen^.portRect:
[Top:0 Left:0 Bottom:13 Right:16]
offScreen^.portBits.bounds:
[Top:0 Left:-3 Bottom:13 Right:16]
sourceRect:
[Top:0 Left:0 Bottom:13 Right:16]
rowBytes = 4

Figure 2–offscreen Aligned

Many, if not most, Color QuickDraw Macintoshes have video cards that can display one pixel per byte, so one would think that pixel alignment does not apply in these cases, since all pixels are at byte boundaries. This statement is true enough, but there is still another kind of alignment that should be done on these machines. Macintoshes with Color QuickDraw generally have full 32-bit microprocessors, and these microprocessors are at their fastest when they can transfer long words aligned on long-word boundaries in memory.

Modifying the last example so that the off-screen pixel map and the screen are both eight-bits-per-pixel, the pixel at the extreme top left corner of the off-screen pixel map is located at a long-word boundary, because the Macintosh Memory Manager forces it to be located there; however, the pixel at the extreme top left corner of the window is located three bytes away from the previous long-word boundary. No bit shifting is needed, because each pixel takes up a whole byte, but `_CopyBits` does have to take the non-optimum case of copying long words on non-long-word boundaries. This case works fine, but it is not quite as fast as it could be. To keep _CopyBits as fast as possible, make sure pixels in the source and destination pixel maps are aligned on long-word boundaries.

Since 1984, Macintosh programmers have been told that rowBytes must be even. That is still true, but to allow _CopyBits to copy an entire pixel map on long-word boundaries, rowBytes must be a multiple of four so that every line in a pixel map begins on a long-word boundary. The following formula can be used to find the minimum rowBytes needed for a pixel map's bounds rectangle with right and left coordinates of bounds.right and bounds.left, and a pixel depth of pixelDepth:

```
rowBytes := ((pixelDepth * (bounds.right - bounds.left) + 31) DIV 32) * 4;
```

Off-screen GWorld support, which was introduced with 32-Bit QuickDraw, can automatically set up a pixel map so that it's properly aligned to any part of the destination pixel map or bit map. You can specify that you want this by passing zero for the pixel depth and passing the rectangle of the destination area in global coordinates. See the 32-Bit QuickDraw Developers' Notes and "Braving Offscreen Worlds" in *develop*, January 1990 for details.

The way that _NewGWorld aligns a GWorld is to set up the off-screen pixel map so that its rowBytes is four bytes wider than one would normally calculate. Four bytes is the maximum amount that any pixel map would have to be realigned at any pixel depth. The bounds rectangle's left coordinate is set to the negative of the left coordinate of the destination rectangle in global coordinates modulo (32 / pixel depth), because this is maximum amount that a pixel map must be shifted to achieve perfect alignment. To build on the earlier example, assume you have a 128-pixel wide, eight-bit deep, off-screen pixel map to copy to a window that is three pixels away from the left edge of an eight-bit color screen.

First, the rowBytes for the off-screen pixel map is set to 131 to allow room for realignment. To align the off-screen pixel map to the on-screen window, the left coordinate of the off-screen bit map's bounds is set to -3 and the right coordinate is still at 128. Notice that the off-screen pixel map's bounds is now 131 pixels wide. Now, the pixels in the off-screen pixel map with a horizontal coordinate of 0 are located three bytes away from the previous long-word boundary. The pixels on the left edge of the window are also located three bytes away from the previous long-word boundary, so _CopyBits can copy long words on long-word boundaries.

If a user moves the window so that it's two pixels from the left edge of the screen, the off-screen pixel map must be realigned. _UpdateGWorld is used to do this. It changes the left coordinate of the off-screen pixel map's bounds rectangle to -2 and then it shifts all the pixels in the off-screen pixel map one pixel to the left. The extra four bytes in each row provide the room for this shifting. (Gives you some new respect for the off-screen support, doesn't it?)

This same discussion applies to any pixel depth, though shallower pixel depths require bit shifting rather than byte shifting. The same principles apply, though. Notice that in a 32-bit deep pixel map, all pixels are aligned on long-word boundaries, so no bit shifting or byte shifting ever needs to be done on one of those. _NewGWorld still adds four to rowBytes even in this case, however.

Modifying the standard test so that the source and destination pixel maps are four bits deep with perfect pixel alignment produces a result of 78 ticks; however, if the destination pixel map is one pixel left of perfect alignment, the result is 228 ticks.

Speed of the Hardware, Of Course

Obviously, the speed of the machine your application is running on affects the speed of _CopyBits. To make _CopyBits as fast as possible, spend a lot of money. However, there is more to the speed of _CopyBits than the speed of the Macintosh itself. When the Macintosh 128K was released, there was only one place for pixel images: main memory. Today, the situation is more complicated. If you have a modular Macintosh, the pixel image for the screen is in the memory of a NuBus™ video card. If you have a Macintosh IIci, you can optionally abandon the NuBus video card and use on-board video which takes up part of main memory. If you have an 8•24 GC card with enough memory, the pixel images can be cached in the card's memory along with the screen's pixel image.

All of these different locations have different access speeds, and that can affect the speed of _CopyBits. Additionally, different Macintoshes have different RAM access speeds. The Macintosh II, IIx, IIcx, and SE/30 have faster RAM than the Macintosh Plus or SE. The Macintosh IIci RAM access speed is faster still, and the Macintosh IIfx has faster RAM access than the IIci. Different video cards have different access speeds. The IIci has a cache card option which can vastly speed up on-board video RAM access speed. Third-party video cards that work in the Processor Direct Slot of the Macintosh SE and SE/30 have their own speed characteristics as well.

There can also be a speed cost for crossing the different areas. If _CopyBits copies between main memory and a NuBus video card, the image data has to be transferred across NuBus. NuBus is a speed bottleneck, so copying an image across NuBus is slower than copying the image from one part of the screen to another or copying from one part of main memory to another. Modifying the standard test to create two windows and two off-screen pixel maps—all eight bits deep with the standard color table then doing every combination of copying between off-screens, between windows, and between off-screens and windows produces the following results:

Off-screen to off-screen:	147
Screen to screen:	188
Off-screen to screen:	204
Screen to off-screen:	201

Performing the standard test on a Macintosh IIfx running System Software 6.0.5 with an Apple Extended Video Card yields a result of 153 ticks, which is not too shabby considering that the transfer is still going through NuBus.

Depth of Pixel Maps

This factor is pretty obvious and is sort of similar to the effect of the dimensions of the copied area: the more bits per pixel there are in the pixel map to copy, the more memory that _CopyBits has to move and the longer it takes to get the job done, assuming that the source and destination pixel maps have the same depth. To make _CopyBits as fast as possible, make sure the pixel maps are as shallow as possible.

If _CopyBits has to copy to a pixel map that has a different depth from the source pixel map, the relationship between speed and depth becomes more complicated. There is a tradeoff between the time taken to change the depth of an image and the absolute amount of data that has to be processed. Copying from a 1-bit deep pixel map to a 32-bit deep pixel map is not that slow because the amount of image data in the 1-bit deep pixel map is so small.

Modifying the standard test to transfer a four-bit deep pixel map to another four-bit deep pixel map produces a result of 78 ticks.

Color Mapping

Color QuickDraw expects a color table attached to every indexed pixel map. Color tables specify what color each pixel value in the pixel map represents. When an application calls _CopyBits to copy a pixel map into another pixel map, _CopyBits reproduces the colors of the image in the source pixel map as closely as possible—even if the colors available in the destination pixel map are different than those available in the source pixel map. This reproduction is done through a process called "color mapping."

When color mapping is done, the source pixel values are transformed into RGBColor records using the source pixel map's color table. These RGBColor records are passed to _Color2Index which finds the pixel values of the closest available colors in the current GDevice pixel map's color table. This same process is done when the source and destination pixel maps have differing depths. The color table attached to the destination pixel map is not used in color mapping. The colors available in the current GDevice pixel map's color table are used instead. So, the destination pixel map must have the same colors for the same pixel values as the current GDevice. Otherwise, the resulting image in the destination pixel map gets the wrong colors. See *Inside Macintosh*, Volume V-141, The Color Manager, for a description of _Color2Index. It's also helpful to read the "Inverse Tables" section in the same chapter on page V-137.

Now, if the source color table contains virtually the same colors for the same pixel values as the current GDevice pixel map's color table, then any particular pixel value has the same color regardless of whether it is in the source or destination pixel map. In this case, color mapping is a waste of time, because the pixels can be copied directly from the source pixel map to the destination pixel map without a loss of color fidelity. _CopyBits takes advantage of this special case to yield some big speed improvements. How is this special case detected? Before this question is answered, it's useful to understand how Color QuickDraw uses color tables.

The ctSeed Field

The first field in a color table is the ctSeed field. This LongInt can be thought of as the color table's version of the scrapCount field of the desk scrap. Whenever an application calls _ZeroScrap, the desk scrap's scrapCount is changed. An application can tell that the desk scrap has changed by checking to see if the scrapCount has changed. Similarly, whenever the contents of a color table are changed in any way, the ctSeed field should be changed to indicate to anyone using that color table that it has been modified.

Additionally, Color QuickDraw often uses the ctSeed as a fast check for color table equality. If two color tables have the same ctSeed, then Color QuickDraw often assumes that their contents are equivalent.

After creating a new color table, an application has to get a valid value for the ctSeed field, and it can do so with the _GetCTSeed routine. This routine generates a valid ctSeed value suitable for a new color table. See *Inside Macintosh*, Volume V-143, The Color Manager, for a description of _GetCTSeed.

System Software 7.0 and 32-Bit QuickDraw each offer a routine called _CTabChanged which should be called after a color table is modified. It takes a handle to the changed color table as a parameter. If the _CTabChanged routine is not available, then the application should instead change ctSeed to a different valid value by calling _GetCTSeed and assigning the result to ctSeed, just like it's done when the application creates a new color table. You must use either one of these methods to tell Color QuickDraw that the color table has changed, or else the modified color table could be confused with the old color table, or with some other color table—this is especially critical if an 8•24 GC card is being used. See the 32-Bit QuickDraw Developers' Notes for details about the _CTabChanged routine.

The ctFlags Field

The ctFlags field is used as a set of flags that indicate some characteristics of the color table. Currently, only the top two bits of ctFlags are of any interest to developers. The most significant bit of ctFlags (bit 15) indicates whether the color table is a sequential color table or an indexed color table. Bit 14 indicates that the color table is a special kind of sequential table if it is set. In these kinds of color tables, the value fields indicate a palette entry in the destination window's palette. See the Palette Manager section of the 32-Bit QuickDraw Developers' Notes for a discussion about this capability.

Sequential Color Tables

If bit 15 of ctFlags is set, the color table is a sequential color table. Sequential color tables are usually found attached to GDevice pixel maps and to GWorld pixel maps.

In sequential color tables, the position of each color in the color table indicates the pixel value to which it corresponds. For example, the fifth entry in a sequential color table always has a pixel value of four (pixel values start at zero). The value field of each ColorSpec is not defined in sequential color tables, though they are used in color tables for screen GDevice records to indicate that a particular color is reserved, protected, or both.

Indexed Color Tables

If bit 15 and 14 of ctFlags are clear, the color table is an indexed color table. In indexed color tables, the value field of each ColorSpec indicates the pixel value of the RGB in that ColorSpec. For example, if the fifth ColorSpec in the color table has a value field containing 10, then that color has a pixel value of 10, not 4, as it would have been if this were a sequential color table.

Color Mapping or Non-Color Mapping

As noted before, _CopyBits can detect whether it has to do color mapping or not, so that it can take advantage of the speed benefits of no color mapping if possible. How is this done? First, _CopyBits checks to see if the ctSeed field of the source and destination color tables are the same and if the source and destination pixel maps have the same depths. If both of these conditions are true, then _CopyBits assumes that the two color tables are identical and it just copies the pixels directly without color mapping. If the ctSeed fields are different, _CopyBits checks manually through all of the colors in the source pixel map's color table map to see if they map to the same pixel values in the current GDevice pixel map's color table as they do in their own color table. If they do, then _CopyBits again takes the fast case.

So to keep _CopyBits as fast as possible, make sure that the source and destination color tables have virtually the same colors for the same pixel values. This applies even if one color table is an indexed color table and the other is a sequential color table, or if the source and destination color tables are both indexed but the order of the ColorSpec records differ.

Modifying the standard test so that the source pixel map has a color table that is the reverse of the standard eight-bit system color table (the grays have low pixel values and the light pinks and yellows have high pixel values) and the destination pixel map has the standard eight-bit system color table produces a result of 470 ticks.

By the way, color tables do not make any sense for direct pixel maps, so this discussion does not apply to them. Direct pixel maps do have a color table attached to them, but they're just there so that an application that assumes that a color table is attached does not bomb.

Scaling

If the source and destination rectangles are the same size, `_CopyBits` has the fairly easy task of just transferring the pixels from the source pixel map to the destination pixel map; however, if the source and destination rectangles are different sizes, `_CopyBits` has to scale the copied image, which slows it down a *lot*. To keep `_CopyBits` as fast as possible, make sure the source and destination rectangles have the exact same dimensions.

Modifying the standard test to copy a 128 by 128 pixel portion of the source pixel map to the whole 256 by 256 pixel window produces a result of 1,159 ticks.

Of Time and Space

Hopefully, this Note makes it a lot clearer to you how to set up a situation in which your `_CopyBits` calls are as fast as your situation allows. It's important to realize that this Note does not cover every single factor that has an influence on the speed of `_CopyBits`. There are many more factors which are just too unpredictable. For example, `_CopyBits` is highly optimized for many special cases, and those optimizations can have a big effect on the speed of the copy. Also, the speed of `_CopyBits` can be affected by interrupt-level tasks. It's up to you to fine tune your programs to your particular situations.

Further Reference:

- *Inside Macintosh*, Volume I, QuickDraw
- *Inside Macintosh*, Volume V, The Color Manager
- *Inside Macintosh*, Volume VI, Color QuickDraw
- Technical Note #163, Adding Color With _CopyBits
- *develop*, January 1990, "Realistic Color for Real-World Applications"
- *develop*, January 1990, "Braving Offscreen GWorlds"
- 32-Bit QuickDraw Developers' Notes (APDA)

NuBus is a trademark of Texas Instruments

#278: _PBClose the Barn Door

Written by: Dave Radcliffe June 1990

This Technical Note discusses the need for Macintosh device drivers to implement `_PBClose`.

Introduction

You may get the idea when implementing device drivers that `_PBClose` is superfluous. After all, if you have a resident driver for a NuBus™ video board, the only time your driver is not needed is if the operating system is going away. It might seem that nothing important can happen after the operating system goes away, so why bother with `_PBClose`? Well, it turns out a lot can happen, and this Note tells you why it is important to implement a `Close` (`_PBClose`) routine.

Transformations

The problem with your driver not being needed when the operating system is going away is that there is more than one way for the operating system to go away. Besides the user choosing Restart or Shutdown from the Special menu, it is possible for the Macintosh operating system to be transformed into an entirely new operating system. This is exactly what happens when A/UX starts.

The A/UX Startup application (Sash in the pre-A/UX 2.0 days) attempts to shut down the Macintosh operating system as much as possible, including closing all drivers, loading A/UX, and starting it running. It does not perform a `RESET`, so if you have not disabled interrupts in your `Close` routine, A/UX may get up and running and find itself receiving interrupts for a board about which it knows nothing—a situation which makes it very unhappy.

In summary, A/UX Startup issues a `_PBClose` call to your driver, so your driver **must** implement a `Close` routine and it **must** accomplish the following (even if you never intend to support your board under A/UX):

- Disable all interrupts for your device.
- Remove your interrupt handler, replacing any changed interrupt vectors.
- Release any private data storage held by your driver.

Further Reference:

- *Inside Macintosh*, Volumes II, IV, & V, The Device Manager
- *Designing Cards and Drivers for the Macintosh Family*, Second Edition

NuBus is a trademark of Texas Instruments.

#279: 'LDEF' Madness

Written by: Byron “scapegoat” Han & Alex Kazim

June 1990

This Technical Note uncovers a problem with writing Pascal list definition procedures and two (yes, count ’em, two) different methods to work around it.

The Hook

List definition procedures ('LDEF' resources) are pieces of stand-alone code that specify the behavior of a list (i.e., how items are drawn and highlighted, etc.) You can write these procedures in a high-level language or in assembly-language, and they have an entry point with the following calling convention:

```
PROCEDURE MyList(lMessage: INTEGER; lSelect: BOOLEAN; lRect: Rect; lCell: Cell;
                 lDataOffset, lDataLen: INTEGER; lHandle: ListHandle);
```

Note that the lRect parameter is a structure greater than four bytes in length, so you must pass a pointer to it. If you write the list definition procedure in a language like Pascal, the rectangle pointed to by lRect is copied into a safe, locally modifiable place.

The Line

When an application calls LNew, the List Manager performs its own initialization prior to calling the list definition procedure with the lInitMsg message. Note that since the initialization of the list does not deal with any cells directly, the lSelect, lRect, lCell, lDataOffset, and lDataLen parameters are supposed to be ignored by the list definition procedure when dealing with the lInitMsg message.

The Sinker

The problem is that the List Manager stuffs garbage into these parameters. Therefore, when the list definition procedure tries to copy the cell rectangle into a local copy, the pointer to the cell rectangle has a chance of being odd, which causes an address error on 68000-based machines, and it is likely to generate a bus error on all other machines.

Solution A

A simple assembly-language header for the list definition procedure to even out the cell rectangle pointer for the `lInitMsg` message can fix the problem:

```
MainLDEF            MAIN    EXPORT
                    IMPORT  MyLDEF
; Stack Frame definition
LHandle             EQU     8                  ; Handle to list data record
LDataLen            EQU     LHandle+4          ; length of data
LDataOffset         EQU     LDataLen+2         ; offset to data
LCell               EQU     LDataOffset+2      ; cell that was hit
LRect               EQU     LCell+4            ; rect to draw in
LSelect             EQU     LRect+4            ; 1=selected, 0=not selected
LMessage            EQU     LSelect+2          ; 0=Init, 1=Draw, 2=Hilite, 3=Close
LParamSize          EQU     LMessage+2-8       ; # of bytes of parameters
                    BRA.S   @0                 ; enter here
; standard header
                    DC.W    0                  ; flags word
                    DC.B    'LDEF'             ; type
                    DC.W    0                  ; LDEF resource ID
                    DC.W    0                  ; version
@0                  LINK    A6,#0
                    MOVE.W  LMessage(A6),D0    ; get the message
                    CMP.W   #lInitMsg,D0
                    BNE.S   @1                 ; not initialization message
                    MOVE.L  #0,LRect(A6);      ; guarantee that this is even
@1                  UNLK    A6
                    JMP     MyLDEF
                    RTS
                    END
```

The code fragment guarantees that when the list definition procedure tries to copy the `lRect` parameter to a safe place, a bus error does not occur.

Solution B

A simpler solution is to declare the entry point to your Pascal `'LDEF'` to be the following:

```
    PROCEDURE MyList(lMessage: INTEGER; lSelect: BOOLEAN; VAR lRect: Rect; lCell: Cell;
                     lDataOffset, lDataLen: INTEGER; lHandle: ListHandle);
```

This revised declaration disables the Pascal compiler's automatic copying of the rectangle data; you need to take care not to modify the cell rectangle passed in `lRect`.

Safe Family Experience

Writing list definition procedures can be a rich and rewarding experience and is a great thing to do on a Saturday night. With a little bit of assembly-language glue, it can even be a safe family experience too.

Further Reference:

- *Inside Macintosh*, Volume IV, The List Manager Package
- Technical Note #110, MPW: Writing Stand-Alone Code

#280: "Bugs In MacApp? Yes, But I Love It!"

Written by: Keith Rollin, Norbert Lindenberg, the MacApp Engineers, and You August 1990

This Technical Note describes latest information about bugs or unexpected "features" in MacApp. Where possible, solutions and fixes are noted. DTS intends this Note to be a complete list of all known bugs in MacApp and will update it as old bugs are fixed or new ones appear. If you have encountered a bug or unexpected feature which is not described here, be sure to let DTS know. Specific code examples and suggested fixes are useful.

This version of the Note reflects the state of MacApp 2.0, released March 22, 1990. Unless otherwise noted, all bugs with listed fixes will be fixed in the next version of MacApp. A script to apply many of the fixes listed here can be found on the Developer CD Series, AppleLink (Developer Support:Developer Services:Macintosh Development Tool Discussions:MacApp Discussion), and the Apple FTP site.

The MacApp Management would like to note that MacApp is a high velocity ride with many twists and turns (all alike). Please keep your hands inside at all times.

There are 107,737 lines of Object Pascal, C++, Assembly, and Rez code that go into the MacApp Library and Build system. As such, it is inevitable that a few bugs creep in. The purpose of this Note is to inform you of these bugs, not to scare you away from MacApp. There are dozens of commercially available programs that lead normal everyday lives which are built on top of MacApp as it stands today. Most of the bugs listed here won't show up in regular use (at least, they didn't in our test programs), so they may not affect you. If they do, you can use the fixes or solutions identified here ("Fixes" are intended to be applied directly to the MacApp source, while "solutions" identify techniques to override or avoid a method to resolve the problem).

MacApp.Lib Bugs

TApplication

1. When being suspended in MultiFinder, MacApp commits command objects which affect the clipboard, rather than checking if the scrap has changed when switching back in.

   Solution: Not yet determined. This is an area of serious consideration for the next version of MacApp.

2. MacApp should hide the clipboard window on a suspend event and redisplay it on a resume event.

   Solution: Override the `TApplication` methods `AboutToLoseControl` and `RegainControl`. `AboutToLoseControl` should remember whether or not the clipboard window is currently open and call `gClipWindow.Close` if it is. `RegainControl` should look at the state of the clipboard window saved by `AboutToLoseControl`, and call `gClipWindow.Open` if the window needs to be reshown.

3. There are problems with the value of the `mouseDidMove` parameter to `TCommand` when called by `TApplication.TrackMouse`. When the `TrackPhase` is `trackPress`, `TCommand.TrackMouse` is called with `mouseDidMove` set to `TRUE` even though the mouse hasn't had a chance to move. When the `TrackPhase` is `trackMove`, `mouseDidMove` is `FALSE` whenever the mouse moves back inside the hysteresis range. When the `TrackPhase` is `trackRelease`, `mouseDidMove` is `TRUE` even if the mouse never moved.

Fix: In `TApplication.TrackMouse` (file UMacApp.TApplication.p):

- The first call to `TrackOnce` should read:

```
TrackOnce(trackPress, FALSE);
```

- The assignment of `didMove` should read:

```
didMove := movedOnce & (NOT EqualVPt(previousPoint, theMouse)).
```

- The last call to `TrackOnce` should read:

```
TrackOnce(trackRelease, didMove);
```

Once those changes have been applied, the parts of MacApp that assume `mouseDidMove = TRUE` when `aTrackPhase = TrackPress` need to be updated. In the methods `TCellSelectCommand.TrackMouse` and `TRCSelectCommand.TrackMouse` (file UGridView.incl.p), replace:

```
IF mouseDidMove THEN
```

With:

```
IF mouseDidMove | (aTrackPhase = TrackPress) THEN
```

4. With these changes, it is possible to experience some feedback problems. For example, when resizing the column widths in a spreadsheet, Calc draws the initial vertical line, waits until the mouse moved outside the hysteresis range, and then, before drawing the vertical line in its new location, erases the old vertical line in the wrong place. This leaves two vertical lines on the screen as garbage.

Fix: In UMacApp.TApplication.p, replace the fourth occurrence of:

```
previousPoint := theMouse;
```

With:

```
IF didMove THEN
    previousPoint := theMouse;
```

5. After performing mouse tracking, `TApplication` calls your `TCommand.DoIt` method, even if the user hasn't moved outside the hysteresis range.

   Fix: Add the following lines after `TrackMouse := tracker;`

```
IF NOT movedOnce THEN
    BEGIN
    TrackMouse := NIL;
    tracker.Free;
    END;
```

TCommand (including subclasses)

1. `TCommand.TrackMouse` fails when `SELF.fView` is `NIL` and `TrackPhase` is `trackRelease`. This is because there is an attempt to call a method of `fView` without first checking if `fView` is `NIL`.

   Fix: In `TCommand.TrackMouse` (file UMacApp.TCommand.p), replace:

```
IF (aTrackPhase = trackRelease) & (NOT fView.ContainsMouse(nextPoint)) THEN
    TrackMouse := NIL
ELSE
    TrackMouse := SELF;
END;
```

   With:

```
IF (aTrackPhase = trackRelease) & ((fView = NIL)
  | (NOT fView.ContainsMouse(nextPoint))) THEN
    TrackMouse := NIL
ELSE
    TrackMouse := SELF;
END;
```

2. If a failure occurs in `TDocument.Revert`, `TRevertDocCommand.DoIt` tries to show the reverted document. This is the correct thing to do if the user canceled out of the revert if a silent failure is signaled (this could happen in `DiskFileChanged`). However if a real error occurred, you cannot leave the document open; you definitively must close it. Otherwise the application may bomb in the next operation involving the document (e.g., the next screen refresh).

   We have to distinguish three classes of errors:

   1) the user canceled out of the operation in `CheckDiskFile`,
   2) a real error was discovered in `CheckDiskFile`,
   3) a real error occurred during rebuilding the document in `DoInitialState` or `ReadFromFile`.

   In the first and second cases, the memory-resident version of the document has not been changed when you reach `HdlRevertCmd`. In the third case, the document may be severely damaged. Therefore, in the first two cases there is no need to call `ShowReverted` (it doesn't hurt either), while in the third case you **must** close the document.

   Case one is easy to recognize (`error = 0`), but for the second and third cases, `error <> 0`. To distinguish between them, you can pull a trick: you know that the Revert menu item is only enabled if `fDocument.fChangeCount` is greater than zero. Therefore, you move `SetChangeCount(0)` in `TDocument.Revert` before any operation that can clobber the document (i.e., before the call to `FreeData`). This way, you can distinguish between the second and third cases in `HdlRevertCmd` by checking `fChangeCount`.

Fix: Change the failure handling procedure in `TRevertDocCommand.DoIt` (file UMacApp.TDocument.p) to:

```
PROCEDURE HdlRevertCmd(error: OSErr; message: LONGINT);
    BEGIN
    {Check whether the document has already been clobbered }
    IF fChangedDocument.GetChangeCount = 0 THEN
        fChangedDocument.Close  {remove the debris left by fChangedDocument}
    END;
```

In `TDocument.Revert`, move the line

```
SetChangeCount(0);
```

before the line

```
FreeData;
```

3. It is potentially problematic having Page Setup as an undoable command, since the view and printer driver context can change. An example of this is shown with the following steps:

    1) Launch any MacApp application.
    2) Access the Page Setup command from the File menu.
    3) Take notice of which printer driver is currently being used and make a change to the dialog (i.e., switch to "landscape" printing), click on the OK button.
    4) Access the Chooser desk accessory and change to a different printer driver.
    5) Now select Redo Page Setup Changes from the Edit menu, then select Undo Page Setup Changes.
    6) Open the Page Setup dialog from the File menu and notice that the "landscape" printing icon is no longer highlighted.
    7) Although the Page Setup dialog is unaffected by Undo and Redo, the document itself is affected, as it prints out in landscape mode, while the Page Setup dialog shows it is in non-landscape mode.

Solution: Apple does not yet have a complete solution to this. If it bothers you, you could modify `IPrintStyleChangeCommand` to make page setup non-undoable.

TControl

1. MacApp's subclasses of `TControl` (defined in the file UDialog.inc1.p) don't pass on their `itsDocument` parameter to the `INHERITED IRes` method. This causes the `fDocument` field to get initialized with `NIL` rather than the `TDocument` reference.

   Solution: You can override the `IRes` method of your own controls to do an `INHERITED IRes` and then set the `fDocument` field to `itsDocument`:

```
PROCEDURE TMyButton.IRes(itsDocument: TDocument;
                         itsSuperView: TView;
                         VAR itsParams: Ptr); OVERRIDE;

    BEGIN
    INHERITED IRes(itsDocument, itsSuperView, itsParams);
    fDocument := itsDocument;
    END;
```

   Then register your class in your `IYourApplication` method so that all Button references in your `'view'` resources result in `TMyButtons` being created, rather than `TButtons`:

```
RegisterStdType('TMyButton', kStdButton);
```

   However, this solution does not work if you depend on these views appearing in the document's `fViewList`.

   Fix: Replace the calls to `INHERITED IRes` in the `IRes` methods of subclasses of `TControl`:

```
INHERITED IRes(NIL, itsSuperView, itsParams);
```

   With:

```
INHERITED IRes(itsDocument, itsSuperView, itsParams);
```

2. Printing disabled controls, especially buttons, results in a gray pattern being printed over the control. This is not a bug in MacApp, but rather a limitation of the LaserWriter. The LaserWriter driver doesn't respect all QuickDraw transfer modes, including the one used to draw the grey text.

   Solution: Not yet determined. It may involve imaging the button into an off-screen bitmap, and then copying it to its destination.

TCtlMgr

1. `TCtlMgr.Draw` has a `WITH` statement that's too long. A field of `fCMgrControl` is set after memory has moved, overwriting four bytes of memory.

   Fix: In `TCtlMgr.Draw` (file UMacApp.TControls.p), replace:

```
WITH fCMgrControl^^ DO
    BEGIN
    savedOwner := contrlOwner;
    contrlOwner := WindowPtr(thePort);

    PenNormal;                {NECESSARY?}

    IF qNeedsROM128K | gConfiguration.hasROM128K THEN
        Draw1Control(fCMgrControl)
    ELSE
        BEGIN
        SetCMgrVisibility(FALSE);             { Force ShowControl to redraw }
        ShowControl(fCMgrControl);
        END;

    contrlOwner := savedOwner;
    END;
```

   With:

```
WITH fCMgrControl^^ DO
    BEGIN
    savedOwner := contrlOwner;
    contrlOwner := WindowPtr(thePort);
    END;                      { MEB moved up from below }
PenNormal;                    {NECESSARY?}

IF qNeedsROM128K | gConfiguration.hasROM128K THEN
    Draw1Control(fCMgrControl)
ELSE
    BEGIN
    SetCMgrVisibility(FALSE);                 { Force ShowControl to redraw }
    ShowControl(fCMgrControl);
    END;

fCMgrControl^^.contrlOwner := savedOwner;   { MEB used to be in WITH }
```

TDeskScrapView

1. If an external application puts both "PICT" and "TEXT" data in the desk scrap, `TDeskScrapView` gets confused and tries to display the "PICT" data as text.

   Fix: Reverse the test in `TDeskScrapView.CheckScrapContents` (file UMacApp.TDeskScrapView.p) so that it looks for "PICT" first and then "TEXT":

```
fHavePicture := LookForScrapType('PICT');
fHaveText := LookForScrapType('TEXT');
```

TDialogView

1. `TDialogView` calls `DoChoice` on a disabled button as the result of a key press. If one disables the default button and presses Return, for example, the button's `DoChoice` method still gets called.

   Fix: The following lines of code appear in `TDialogView.DoCommandKey` and in `TDialogView.DoKeyCommand` (file UDialog.incl.p):

```
IF cancelView.IsViewEnabled THEN
    TControl(cancelView).Flash;
TControl(cancelView).DoChoice(cancelView,TControl(cancelView).fDefChoice);
```

   Replace them with:

```
IF cancelView.IsViewEnabled THEN
    BEGIN
    TControl(cancelView).Flash;
    TControl(cancelView).DoChoice(cancelView,
                                  TControl(cancelView).fDefChoice);
    END;
```

   Additionally, in `TDialogView.DoKeyCommand`, replace:

```
IF defaultView.IsViewEnabled THEN
    TControl(defaultView).Flash;
TControl(defaultView).DoChoice(defaultView,TControl(defaultView).fDefChoice);
```

   Replace them with:

```
IF defaultView.IsViewEnabled THEN
    BEGIN
    TControl(defaultView).Flash;
    TControl(defaultView).DoChoice(defaultView,
                                   TControl(defaultView).fDefChoice);
    END;
```

   Solution: You can do this as an `OVERRIDE` if you hesitate to change MacApp.

TDocument

1. `TDocument.Save` fails if you lock a file after opening it with read and write access and then try to save. The file is closed and `fDataRefNum` and `fRsrcRefNum` contain their old (and now invalid) values.

   Solution: Not yet determined.

2. If `GetFileInfo` returns a result other than `noErr`, `TDocument.DiskFileChanged` maps it to `errFileChanged`, because there is no check for (`err = noErr`) in the `ELSE IF` branch. The resulting alert is misleading, as the file may also have been renamed, deleted, or the file server may have gone offline.

Fix: The error checking code in `TDocument.DiskFileChanged` (file UMacApp.TDocument.p) should look like:

```
err := GetFileInfo(fTitle^^, fVolRefNum, pb);
...
IF (err = noErr) THEN
    IF checkType & (pb.ioFlFndrInfo.fdType <> fFileType) THEN
        err := errFTypeChanged
    ELSE IF pb.ioFlMdDat <> fModDate THEN
        err := errFileChanged;
DiskFileChanged := err;
```

3. It is not possible to use the Pascal built-in filing function `Close` from within a `TDocument` method because the Object Pascal scoping rules always associate the name `Close` with `TDocument.Close`.

Solution: It is likely that Apple will change the name in the future. After all, there are three distinct objects that implement a `Close` method, none of which have any relation to another; something like that needs to be cleaned up. In the meantime, you could make a global routine `MyClose` that would be a wrapper for the `Close` routine.

TEditText

1. If the first or only `TEditText` in a dialog has auto-wrap turned on and is not initially selected, tabbing to it after opening the window selects it, but the selection is not visible until the window is refreshed. This does not occur if auto-wrap is turned off for that `TEditText`.

Solution: Not yet determined.

TGridView

1. Attempting to select a `TGridViewcell` for which `CanSelectCell` returns `FALSE` causes the current selection to be deselected.

Solution: Override `TGridView.DoMouseCommand to` call `IdentifyPoint`. If a valid cell is returned, call `CanSelectCell`. If it returns `TRUE`, call `INHERITED DoMouseCommand`. This inhibits all tracking if the user initially clicks in a disabled cell.

Fix: Replace the following line in `TCellSelectCommand.TrackMouse` (file UGridView.inc1.p):

```
IF LONGINT(clickedCell) <> LONGINT(fPrevCell)
```

With:

```
IF (LONGINT(clickedCell) <> LONGINT(fPrevCell))
  & fGridView.CanSelectCell(clickedCell)
```

2. Neither `TGridView.IRes` nor `TGridView.IGridView` follow standard initialization protocol. They each attempt to create their supporting regions and objects before calling their superclass' initialization method, without having placed the object in a freeable state first.

Solution: (for `IGridView`): Override `TGridView.IGridView` as follows:

```
PROCEDURE TMyGridView.IGridView(a long list of parameters); OVERRIDE;
    BEGIN
    fSelections := NIL;
    fHLRegion := NIL;
    fTempSelections := NIL;
    fColWidths := NIL;
    fRowHeights := NIL;
    INHERITED IGridView(xxx);
    ...
    END;
```

Solution: (for `IRes`): In your `TMyGridView.IRes` method, insert the following statements before calling `INHERITED IRes`:

```
fSelections := NIL;
fHLRegion := NIL;
fTempSelections := NIL;
fColWidths := NIL;
fRowHeights := NIL;
```

TIcon

1. `TIcon.SetIcon` does not support switching between color and black and white icons.

Solution: Create your own subclass of `TIcon`, and override `TIcon.SetIcon`. Use something like this:

```
PROCEDURE TMyIcon.SetIcon(theIcon: Handle;
                         redraw: BOOLEAN); OVERRIDE;

    BEGIN
    ReleaseIcon;

    IF GetHandleSize(theIcon) <> 128 THEN
        fPreferColor:= TRUE
    ELSE
        fPreferColor:= FALSE;

    fDataHandle := theIcon;
    IF redraw THEN
        ForceRedraw;
    END;
```

TList

1. If you have a `TList` subclass with a `String` instance variable, it is not possible to use the Pascal string built-in function `Delete` on it because the Object Pascal scoping rules always associate the name Delete with `TList.Delete`.

Solution: Apple will change the name in the future. In the meantime, you could make a global routine `MyDelete` that would be a wrapper for the string `Delete` routine.

TNumberText

1. When the length of the text in a `TNumberText` instance is 0, `GetValue` returns 0, and `Validate` returns `kValidValue`. The value is not checked against `fMinimum` or `fMaximum`, so your application may be fed with a value it is not prepared to handle.

   Fix: Ideas for solutions or fixes are outlined in the comment in `TNumberText.Validate` (file UDialog.incl.p).

TPopup

1. Calling `TPopup.DrawPopupBox` results in a bus error if the popup menu item has no text (`_GetItem` returns `theItem = ""`, which can happen, for example, if the menu resource has no items in it), or if the popup's menu rectangle is narrow: right - left <= 15. This is because `DrawPopupBox` never takes into account popups that have no text.

   Solution: Make sure that the menu you install has at least one item in it, and make sure that your popup's width is least 16 pixels.

   Fix: In `TPopup.DrawPopupBox` (file UDialog.incl.p), replace:

```
IF SectRect(area, menuRect, colorRect) THEN
    BEGIN
    wid := (right - left) - (leftSlop + rightSlop);
    …
```

   With:

```
IF SectRect(area, menuRect, colorRect) & (theItem <> '') THEN
    BEGIN
    wid := Max(kMinWidth, (right - left) - (leftSlop + rightSlop));
    …
```

   Where `kMinWidth` could reasonably be any value from zero to eight.

2. If the menu background color is not white, `TPopup.DrawPopupBox` leaves a one-pixel wide white line along the bottom right edge of the update area (try it with DemoDialogs by partially covering the colored popup with a window and then uncovering it).

   Fix: In `TPopup.DrawPopupBox` (file UDialog.incl.p), replace:

```
WITH colorRect DO
    BEGIN
    right := right - 1;
    bottom := bottom - 1;
    END;
```

   With:

```
WITH colorRect DO
    BEGIN
    right := MIN(right, menuRect.right - 1);
    bottom := MIN(bottom, menuRect.bottom - 1);
    END;
```

3. TPopup.ReleasePopup should call _DisposeMenu rather than _ReleaseResource. Because TPopup.SetPopup calls _DetachResource on theMenu, fMenuHandle is no longer recognized as referring to a resource, and _ReleaseResource fails.

Fix: In TPopup.ReleasePopup (file UDialog.inc1.p), replace:

```
IF fMenuHandle <> NIL THEN
    BEGIN
    HPurge(Handle(fMenuHandle));
    ReleaseResource(Handle(fMenuHandle));
    fMenuHandle := NIL;
    END;
```

With:

```
IF fMenuHandle <> NIL THEN
    BEGIN
    DisposeMenu(fMenuHandle);
    fMenuHandle := NIL;
    END;
```

4. TPopup no longer calls DoChoice if the same item is reselected.

Fix: In TPopup.DoMouseCommand (file UDialog.inc1.p) is the following line:

```
IF (HiWord(result) <> 0) & (newChoice <> fCurrentItem) THEN
```

Remove the "& (newChoice <> fCurrentItem)" part.

5. TPopup.SetCurrentItem neither restores the port colors correctly nor uses the right rectangle to obtain the menu colors for the popup box.

Fix: In TPopup.SetCurrentItem (file UDialog.inc1.p), replace:

```
IF redraw & Focus & IsVisible THEN
    BEGIN
    GetQDExtent(menuRect);
    GetMenuColors(menuRect, fMenuID, item, newFColor, newBkColor);
    SetIfColor(newFColor); SetIfBkColor(newBkColor);
    DrawPopupBox(menuRect);
    END;
```

With:

```
IF redraw & Focus & IsVisible THEN
    BEGIN
    GetIfColor(oldFColor); GetIfBkColor(oldBkColor);
    CalcMenuRect(menuRect);

    GetMenuColors(menuRect, fMenuID, fCurrentItem,
                  newFColor, newBkColor);
    SetIfColor(newFColor); SetIfBkColor(newBkColor);
    DrawPopupBox(menuRect);

    { Reset colors to their original state }
    SetIfColor(oldFColor); SetIfBkColor(oldBkColor);
    END;
```

TScroller

1. TScroller.RevealRect doesn't call INHERITED RevealRect. This has implications in situations where you have nested scrollers. If, for example, you run DemoDialogs, select the first menu item, press the Tab key, then begin typing, the TEditText item you are modifying is not scrolled into view. This is because while your selection is revealed within the context of the TEditText, the TEditText item itself is not scrolled into view.

   Fix: Add an INHERITED RevealRect call to TScroller.RevealRect (file UMacApp.TScroller.p):

```
PROCEDURE TScroller.RevealRect(...);
    BEGIN
    ...
    ScrollBy(delta.h, delta.v, redraw);
    INHERITED RevealRect(rectToReveal, minToSee, redraw); { add this line }
    END;
```

TStdPrintHandler

1. An extra blank page is printed if TStdPrintHandler.fFixedSizePages = FALSE and fSizeDeterminer = sizeFillPages. This is because TView.ComputeSize computes the view's size as a multiple of the printable page size for sizeFillPages, ignoring that the view need not use the full size of each page.

   Solution: Always set both Boolean components of fFixedSizePages to TRUE. These are initialized from the last two parameters you pass to IStdPrintHandler.

   Solution: Use fSizeDeterminer = sizeVariable.

2. Simply using the naked DIV operator for scaling theMargins in TStdPrintHandler.CheckPrinter introduces rounding errors. These errors may be disturbing if you need precise control over the margins used for printing.

   Fix: Insert the following local procedure in TStdPrintHandler.CheckPrinter (file UPrinting.inc1.p):

```
FUNCTION ScaleInteger(theValue, theMultiplier, theDivisor: Integer): Integer;
    VAR
        intermediate: Longint;
    BEGIN
        intermediate := IntMultiply(theValue, theMultiplier);
        IF intermediate >= 0 THEN
            theProduct := theProduct + Abs(theDivisor) div 2
        ELSE
            intermediate := intermediate - ABS(theDivisor) div 2;
        ScaleInteger := intermediate DIV theDivisor;
    END;
```

In the implementation of `TStdPrintHandler.CheckPrinter`, replace the lines:

```
SetRect(theMargins, IntMultiply(theMargins.left, h) DIV oldMarginRes.h,
                    IntMultiply(theMargins.top, v) DIV oldMarginRes.v,
                    IntMultiply(theMargins.right, h) DIV oldMarginRes.h,
                    IntMultiply(theMargins.bottom, v) DIV oldMarginRes.v);
```

With:

```
SetRect(theMargins,
        ScaleInteger(theMargins.left, fMarginRes.h, oldMarginRes.h),
        ScaleInteger(theMargins.top, fMarginRes.v, oldMarginRes.v),
        ScaleInteger(theMargins.right, fMarginRes.h, oldMarginRes.h),
        ScaleInteger(theMargins.bottom, fMarginRes.v, oldMarginRes.v));
```

3. `TStdPrintHandler.CheckPrinter` calculates `fMarginRes` incorrectly for scaled printing. It does not take into account any scaling factors imposed by the user in the `_PrStlDialog` dialog box.

Temporary Fix: Use the following until Apple can come up with something better. Note that this fix relies on the undocumented fields `prStl.iPageV` and `prStl.iPageH`. Additionally, it implements a dubious technique that gets around the assumption that any printer supporting landscape printing also supports `_PrGeneral`, which is not always the case (e.g., AppleLink LinkSaver). Therefore, this fix is considered temporary. You should already have applied the fix to the second bug in the TStdPrinthandler section.

Insert the following local procedure into `TStdPrintHandler.CheckPrinter` (file UPrinting.incl.p):

```
PROCEDURE AdjustMarginRes;

    VAR
        getRotationBlock: TGetRotnBlk;

    BEGIN
         WITH getRotationBlock DO BEGIN
             iOpCode := getRotnOp;
             lReserved := 0;
             hPrint := THPrint(fHPrint);
             bXtra := 0;
         END;
         PrGeneral(@getRotationBlock);
         IF (PrError <> noErr) | (getRotationBlock.iError <>
           noErr) THEN BEGIN
             WITH fPageAreas.thePaper DO
                 getRotationBlock.fLandscape := right - left >
                                                bottom - top;
             PrSetError(noErr); { clear print error - Printing
                                  Manager won't do it }
         END;
```

```
            WITH fPageAreas.thePaper, fMarginRes,
              THPrint(fHPrint)^^ DO BEGIN
                {$PUSH} {$H-} { shut up, dumb compiler! }
                IF getRotationBlock.fLandscape THEN BEGIN
                { the undocumented fields prStl.iPageH seem
                  to unaffected by rotation, so we have to
                  rotate them }
                    fMarginRes.h := ScaleInteger(iPrPgFract,
                                right - left, prStl.iPageV);
                    fMarginRes.v := ScaleInteger(iPrPgFract,
                                bottom - top, prStl.iPageH);
                END
                ELSE BEGIN
                    fMarginRes.h := ScaleInteger(iPrPgFract,
                                right - left, prStl.iPageH);
                    fMarginRes.v := ScaleInteger(iPrPgFract,
                                bottom - top, prStl.iPageV);
                END;
                {$POP}
            END; { WITH }
        END;
```

In TStdPrintHandler.CheckPrinter, replace everything after fPageAreas.thePaper := rPaper and up to, but not including, the statement fPrinterDev := iDev; with the following lines:

```
AdjustMarginRes;
WITH prInfo DO BEGIN
```

4. TStdPrintHandler.DoSetupMenus should not enable menus if MemSpaceIsLow.

Solution: Override TStdPrintHandler.DoSetupMenus with the following:

```
PROCEDURE TMyStdPrintHandler.DoSetupMenus; OVERRIDE;

    VAR
        okToEnable:       Boolean;

    BEGIN
    INHERITED DoSetupMenus;

    okToEnable := NOT MemSpaceIsLow;
    IF gCouldPrint & (fView <> NIL) THEN
        BEGIN
        Enable(cPrint, okToEnable);
        Enable(cPageSetup, okToEnable);
        Enable(cPrintOne, okToEnable);
        END;

    EnableCheck(cShowBreaks, TRUE, fShowBreaks);
    END;
```

TTEView

1. `TTEPasteCommand.ITEPasteCommand` strands an empty handle on the heap if there is “TEXT” in the clipboard with no `'styl'` information.

   Fix: Replace the following lines in `TTEPasteCommand.ITEPasteCommand` (file UTEView.TTEPasteCommand.p):

```
IF newStyleLen > 0 THEN
    BEGIN
    fNewStyles := newStyles;
    { Difference between old and new styles }
    fStylePad := newStyleLen - fStylePad;
    END;
```

   With:

```
IF newStyleLen > 0 THEN
    BEGIN
    fNewStyles := newStyles;
    { Difference between old and new styles }
    fStylePad := newStyleLen - fStylePad;
    END
ELSE
    Handle(newStyles) := DisposeIfHandle(newStyles);
```

2. In `TTEView.CalcRealHeight`, the parameters to `TEGetHeight` are in the wrong order.

   Fix: In `TTEView.CalcRealHeight` (file UTEView.TTEView.p), replace:

```
theHeight := TEGetHeight(0, MAXINT, fHTE);
```

   With:

```
theHeight := TEGetHeight(MAXINT, 0, fHTE);
```

3. In a `TTEView` with non-zero bottom inset, only part of the second line is displayed when text wraps to a new line.

   Solution: Always have a bottom inset of zero.

   Fix: Modify `TTEView.Resize` (file UTEView.TTEView.p) to allow for the bottom inset before doing the resize by inserting the following lines between `oldSize := fSize` and `INHERITED Resize`:

```
IF (fInset.bottom <> 0) | (fSize.v <> height) THEN
    BEGIN
    fSize.v := fSize.v - fInset.bottom;
    IF fSize.b = height THEN
        fsize.v := fSize.v - 1;
    END;
```

4. `TTEView.SynchView` only updates the text if the line heights have changed. It calls `CalcRealHeight`, and if it has not changed, it doesn't do anything. If a program modifies the text directly, it must call `ForceRedraw`. For instance, say that you have a class `TMyTEView` has the following routine:

```
PROCEDURE TMyTEView.TweekText;
    VAR
        myText : TextHandle;
    BEGIN
    myText := ExtractText;
    { do some munging of the text (e.g., search and replace) }
    { make TTEView display changed text }
    RecalcText;
    SynchView(kRedraw);
    { !!! We shouldn't have to force a complete redraw !!! }
    ForceRedraw;
    END;
```

Solution: Call `ForceRedraw` as above, until Apple has a solution. It could be that removing the `fLastHeight <> theHeight` comparison in `SynchView` does the trick, but it may also result in unnecessary updates and flashing.

TView

1. `TEvtHandler.DoCreateViews` doesn't work right if you build your view tree in the "wrong" order (i.e., breadth-first order). If you declare them as a hierarchy of levels, like this:

```
ViewA
ViewB
    SubViewA-1
    SubViewA-2
    SubViewB-1
        SubViewA-1-1
        SubViewA-1-2
```

`DoCreateViews` cannot find `SubViewA-1` when creating `SubViewA-1-1`.

Solution: Declare your views in this order (walking the tree) in the Rez file:

```
ViewA
    SubViewA-1
        SubViewA-1-1
        SubViewA-1-2
    SubViewA-2
ViewB
    SubViewB-1
```

2. `TView.Focus` does not always work correctly in long coordinate situations. When dealing with view systems that stay entirely within QuickDraw's 16-bit coordinate plane, focusing works correctly. However, when dealing with larger view systems, `TView.Focus` does not always correctly switch over to MacApp's 32-bit coordinate system.

Fix: In TView.Focus (file UMacApp.TView.p), replace:

```
IF fSize.vh[vhs] > kMaxCoord THEN
```

With:

```
IF (fSize.vh[vhs] > kMaxCoord) | (ABS(fLocation.vh[vhs]) > kMaxCoord)
  | (ABS(gLongOffset.vh[vhs]) > kMaxCoord) THEN
```

Daring Fix: You can try taking out short coordinate focussing altogether. This solution has not yet been fully tested, so there may be some side effects of which Apple is unaware. In TView.Focus (file UMacApp.TView.p), replace:

```
FOR vhs := v TO h DO
    IF fSize.vh[vhs] > kMaxCoord THEN
        BEGIN
        tempLongOffset := gLongOffset.vh[vhs] - fLocation.vh[vhs];
        relOrigin.vh[vhs] := tempLongOffset MOD kMaxOriginFixup;
        gLongOffset.vh[vhs] := tempLongOffset - relOrigin.vh[vhs];
        END
    ELSE
        BEGIN
        relOrigin.vh[vhs] := gLongOffset.vh[vhs] - fLocation.vh[vhs];
        gLongOffset.vh[vhs] := 0;
        END;
```

With:

```
FOR vhs := v TO h DO
    BEGIN
    tempLongOffset := gLongOffset.vh[vhs] - fLocation.vh[vhs];
    relOrigin.vh[vhs] := tempLongOffset MOD kMaxOriginFixup;
    gLongOffset.vh[vhs] := tempLongOffset - relOrigin.vh[vhs];
    END;
```

3. TView calls _InvalRect and _ValidRect directly. These are Window Manager calls which assume that the current port (thePort) is a window. If thePort is not a window and these calls are made, all sorts of nasty fireworks happen. This bug only appears when a TView is placed in something other than a TWindow and the view calls TView.InvalidRect, TView.InvalidRect, or TView.ValidVRect.

For example, when using a TGridView as a subview of a TMenu, IGridView results in a call to TView.InvalidRect. Since TMenu carries its own GrafPort, the InvalRect on the TMenu GrafPort fails.

Fix: In the file UMacApp.TView.p, rewrite the methods TView.InvalidRect, TView.InvalidVRect, and TView.ValidVRect as shown so they forward up the superview hierarchy. Then add the methods TView.ValidateRect, TWindow.InvalidRect, and TWindow.ValidateRect, also as shown. These routines ensure that the Window Manager calls (_InvalRect and _ValidRect) are actually made only if the root view is a window.

With those changes in place, all calls to _ValidRect in the rest of MacApp should now be calls to TView.ValidateRect. The only methods this affects are TSScrollbar.Activate and TDeskScrapView.Draw.

```
{----------------------------------------------------------------------------}
{$S MAViewRes}

PROCEDURE TView.InvalidRect(r: Rect);

    BEGIN
    { Focusing is strictly _NOT_ required here.  That way we can keep
      transforming all the way out to the root view and not change the
      focus along the way }
    IF (fSuperView <> NIL) THEN
        fSuperView.InvalidRect(r);
    END;

{----------------------------------------------------------------------------}
{$S MAViewRes}

PROCEDURE TView.InvalidVRect(viewRect: VRect);

    VAR
        r:              Rect;

    BEGIN
    IF Focus THEN
        BEGIN
        ViewToQDRect(viewRect, r);
        InvalidRect(r);
        END;
    END;

{----------------------------------------------------------------------------}
{$S MAViewRes}

PROCEDURE TView.ValidateRect(r: Rect);

    BEGIN
    { Focusing is strictly _NOT_ required here.  That way we can keep
      transforming all the way out to the root view and not change the
      focus along the way }
    IF (fSuperView <> NIL) THEN
        fSuperView.ValidateRect(r);
    END;

{----------------------------------------------------------------------------}
{$S MAViewRes}

PROCEDURE TView.ValidVRect(viewRect: VRect);

    VAR
        r:              Rect;

    BEGIN
    IF Focus THEN
        BEGIN
        ViewToQDRect(viewRect, r);
        ValidateRect(r);
        END;
    END;
```

```
{------------------------------------------------------------------------------}
{$S MAWindowRes}

PROCEDURE TWindow.InvalidRect(r: Rect); OVERRIDE;

    VAR
        oldPort:        GrafPtr;

    BEGIN
    GetPort(oldPort);
    SetPort(fWMgrWindow);

    IF IsShown THEN
        IF NOT EmptyRect(r) THEN·
            InvalRect(r);

    SetPort(oldPort);
    END;

{------------------------------------------------------------------------------}
{$S MAWindowRes}

PROCEDURE TWindow.ValidateRect(r: Rect); OVERRIDE;

    VAR
        oldPort:        GrafPtr;

    BEGIN
    GetPort(oldPort);
    SetPort(fWMgrWindow);

    IF IsShown THEN
        IF NOT EmptyRect(r) THEN
            ValidRect(r);

    SetPort(oldPort);
    END;
```

4. When the focus is invalidated during printing, MacApp is not able to restore it properly. For example, you could move a subview during printing because you don't know where it's supposed to go until you need it. When MacApp tries to refocus, the clip region is set to an empty region, and nothing gets printed from that point on.

   Solution: Not yet determined. It's not clear whether MacApp should handle such odd things as moving subviews during printing.

TWindow

1. `TWindow.IWindow` incorrectly sets the `fProcID` field. This can affect the performance of `TWindow.SetResizeLimits` and `TWindow.Zoom` (i.e., they break).

   Fix: In `TWindow.IWindow` (file UMacApp.TWindow.p), replace the following lines:

```
fProcID := GetWRefCon(itsWMgrWindow);
SetWRefcon(itsWMgrWindow, LONGINT(SELF));
```

With:

```
IF TrapExists(_GetWVariant) THEN
    fProcID := GetWVariant(fWmgrWindow)
ELSE
    fProcID := BAND($0F, BSR(LONGINT(WindowPeek(fWmgrWindow)^.windowDefProc),
                             24));
SetWRefcon(itsWMgrWindow, LONGINT(SELF));
```

Of course, anyone not creating their windows and views from resource templates should be horse-whipped anyway.

2. `TWindow.Zoom` grows the window two pixels wider and two pixels taller than it should.

Fix: In `TWindow.Zoom` (file UMacApp.TWindow.p), replace the following two lines :

```
zoomRect.right := zoomRect.left + width - fContDifference.h + 1;
zoomRect.bottom := zoomRect.top + height - fContDifference.v + 1;
```

With:

```
zoomRect.right := zoomRect.left + width - fContDifference.h - 1;
zoomRect.bottom := zoomRect.top + height - fContDifference.v - 1;
```

3. `TWindow.Center` can sometimes move large windows with title bars under the menu bar.

Fix: In `TWindow.Center` (file UMacApp.TWindow.p), replace the following lines:

```
IF forDialog THEN
    { Put it in the top third of the screen }
    top := ((screenSize.v - contentSize.v + fContRgnInset.v) DIV 3) + 20
ELSE
    top := ((screenSize.v - contentSize.v + fContRgnInset.v) DIV 2) + 20;
```

With:

```
IF forDialog THEN
    { Put it in the top third of the screen }
    top := ((screenSize.v - windowsize.v) DIV 3)    { calculate spare area }
          + gMBarHeight                                      { add menu bar }
        { calculate the right offset of content inside the window }
          + ((windowsize.v - contentsize.v + fContRgnInset.v) DIV 2);
ELSE
    top := ((screenSize.v - windowsize.v) DIV 2)    { calculate spare area }
          + gMBarHeight                                      { add menu bar }
        { calculate the right offset of content inside the window }
          + ((windowsize.v - contentsize.v + fContRgnInset.v) DIV 2);
```

Assorted Problems Due to a New TView.Focus Definition

The next items address a class of problems related to the fact that `TView.Focus` is defined to return `TRUE` if a drawing environment can be obtained (e.g., a `GrafPort`). Thus it now returns `TRUE` even if the view is invisible. The various problems are: 1) invisible controls in dialogs accepting mouse-down events and doing things; 2) children of invisible controls being asked to draw or handle a mouse-down event; 3) scroll bars of hidden scrollers appearing; 4) hidden scroll bars of scrollers not appearing; and 5) calls to `IsShown` for an arbitrary view returning incorrect results.

1. `TView.IsShown` contains the following line:

```
IsShown := fShown;           {??? Shouldn't we ask our superview? }
```

It turns out that the answer to this question is yes. There are many problems that occur in MacApp that are caused by views who are themselves not hidden, but whose superviews are. For instance, it is possible for a click to be registered on a view whose superview is hidden. This can cause the previously hidden control to appear.

Fix: In `TView.IsShown` (file UMacApp.TView.p), replace the line above with the following:

```
IF fSuperView <> NIL THEN
    IsShown := fShown & fSuperView.IsShown { By definition a view can't be
                                             shown if its superview isn't
                                             shown }
ELSE
    IsShown := fShown;
```

2. Having `TView.IsShown` reflect the willingness of all its superviews to be shown causes one problem in MacApp. When a `TScroller` creates its scroll bars, it sets the `fShown` field of the `TSScrollBar` to the result of `TScroller.IsShown`. However, at the time a scroller creates its scroll bars, the window they are in is invisible. Its `IsShown` method returns `FALSE`, which is propagated down to the `TScroller`, causing `TScroller.CreateTemplateScrollBar` to initialize `TSScrollBar.fShown` to `FALSE`.

Fix: Cause the `TSScrollBar` to inherit the `fShown` field of its `TScroller` **only**. In `TScroller.CreateTemplateScrollBar` (file UMacApp.TScroller.p), replace:

```
anSScrollBar.fShown := IsShown;
```

With:

```
anSScrollBar.fShown := fShown;
```

3. There is no `TCtlMgr.Show` to control the setting of `fCMgrControl^^.contrlVis`. Neglecting to do so results in certain silly things happening, like an activate event triggering the drawing of your invisible scroll bars.

Fix: Override `TView.Show` with the following version of `TCtlMgr.Show` (file UMacApp.TControls.p). Don't forget to also update the declaration of `TCtlMgr` in UMacApp.p:

```
PROCEDURE TCtlMgr.Show(state, redraw: BOOLEAN);

    VAR
        itsWindow:  TWindow;

    BEGIN
    itsWindow := GetWindow;
    SetCMgrVisibility(state & (itsWindow <> NIL) & itsWindow.fIsActive);
    INHERITED Show(state, redraw);
    END;
```

4. TControl.ContainsMouse needs to call TCtlMgr.IsShown. Otherwise, it's possible for those controls to receive mouse clicks.

Fix: Use the following version of TControl.ContainsMouse (file UMacApp.TControls.p):

```
FUNCTION TControl.ContainsMouse(theMouse: VPoint): BOOLEAN; OVERRIDE;
    VAR
                aRect: Rect;
    BEGIN
    IF IsShown THEN
        BEGIN
        ControlArea(aRect);
        ContainsMouse := PtInRect(VPtToPt(theMouse), aRect);
        END
    ELSE
        ContainsMouse := FALSE;
    END;
```

5. TView.Focus used to return FALSE if the view was invisible. It no longer does this, and many routines in MacApp relying on this behavior now need to check this explicitly:

Fix: The following routines should be modified to check IsShown before calling Focus. Note that the changes to TView.InvalidVRect, TView.InvalidVRect, and TView.ValidRect need not be made if the modifications in the third bug in the TView section have been made.

```
TView.IsViewEnabled                                  (file UMacApp.TView.p)
     IsViewEnabled := fViewEnabled & IsShown;

TGridView.HighlightCells                             (file UGridView.incl.p)
     IF (fromHL <> toHL) & IsShown & Focus THEN

TCtlMgr.WhileFocused                                 (file UMacApp.TControls.p)
TTEView.SynchView                                    (file UTEView.TTEView.p)
     IF redraw & IsShown & Focus THEN

TView.InvalidRect ( see above comment )              (file UMacApp.TView.p )
TView.InvalidVRect ( see above comment )             (file UMacApp.TView.p )
TView.ValidVRect ( see above comment )               (file UMacApp.TView.p )
TGridView.InvalidateSelection                        (file UGridView.incl.p)
TScroller.ScrollDraw                                 (file UMacApp.TScroller.p)
     IF IsShown & Focus THEN

TSScrollBar.Activate                                 (file UMacApp.TControls.p)
     add this check before WhileFocused:
     IF IsShown THEN
```

6. With the changes from bug five in place, a problem appears when a TScroller is resized. The scroller hides its scroll bars, resizes itself, adjusts its scroll bars, and shows them again. AdjustScrollbars potentially asks a scroll bar to invalidate itself. However, at that time, the scroll bar is invisible, thus its contents cannot possibly be wrong, as they have yet to be drawn. It is the scroll bar itself that is wrong, and therefore the contents of its superview (in that rectangle) that must be invalidated.

Fix: To patch the bug, modify the final few lines of `TScroller.Resize` (file UMacApp.TScroller.p):

```
FOR vhs := v TO h DO
    IF sBarWasVisible[vhs] THEN
        BEGIN
        fScrollBars[vhs].SetCMgrVisibility(TRUE);
        fScrollBars[vhs].ForceRedraw; { this is new }
        END;
```

This is not a real fix, this is only a patch. The final fix probably requires modification to `TView.Locate` and `TControl.Resize`.

7. It is possible to select a disabled `TEditText` item by clicking on it, because `TEditText.HandleMouseDown` calls `DoChoice` and selects the item without checking to see if the `TEditText` object is disabled.

Fix: In the nested procedure `TestMouse` of `TView.HandleMouseDown` (file UMacApp.TView.p) is the following line:

```
IF theSubView.ContainsMouse(subViewPt) THEN
```

Replace it with:

```
IF theSubView.IsViewEnabled & theSubView.ContainsMouse(subViewPt) THEN
```

Global Routines and Interfaces

1. In `GetPortTextStyle`, `theTextStyle` should be a `VAR` parameter. The fact that it isn't should tell you how often MacApp itself calls it.

Fix: Modify the interface to `GetPortTextStyle` in the files UMacAppUtilities.p and UMacAppUtilities.inc1.p.

2. `DoRealInitToolBox` should call `_FlushEvents(everyEvent-diskMask-`**`app4Mask`**`)` rather than `_FlushEvents(everyEvent-diskMask-app4Evt)`.

Fix: Change `app4Evt` to `app4Mask` in the `_FlushEvents` call (file UMacAppUtilities.inc1.p).

3. MacApp programs do not draw their menu bars if run on pre-System 6.0 systems. This problem is because of a bug in `InvalidateMenuBar`, where a check made for a System 7.0 feature returns `TRUE` under System 4.2 when it should not.

Fix: Fixing the problem requires a slight change to MacApp. In the file UMenuSetup.inc1.p, there is a routine called `InvalidateMenuBar`. Change the line that looks like the following:

```
IF TrapExists(_MAInvalMenuBar) THEN
```

to look like:

```
IF gConfiguration.systemVersion >= $0600 & TrapExists(_MAInvalMenuBar) THEN
```

4. `TestRecoverHandle` in the file UMacAppUtilities.incl.p is used in debug mode to help test whether a handle is truly a handle. On 68000-class machines, `_HandleZone`, which is called as part of the test, drops you into the MacApp debugger with an address error if the `longint` being tested isn't really a handle. This is not a problem on non-68000 machines.

   Fix: If you are developing your application on a Macintosh Plus or SE and want to compile in debug mode, you need to remove the test for `_RecoverHandle`.

```
FUNCTION TestRecoverHandle(masterPointer: Ptr;
                           h: UNIV Handle): Boolean;

    VAR
        ...

    BEGIN
    IF qNeedsMC68020 | qNeedsMC68030 | (gConfiguration.processor
      <> env68000) THEN
        BEGIN
        <Old body of TestRecoverHandle>
    ELSE
        TestRecoverHandle := TRUE;  {ƒƒƒ Can't really test this ƒƒƒ}
    END;
```

5. Sometimes the high byte of a handle is not zero (for example, `_NewEmptyHandle` can return something like $404856d0). When this occurs, `TestRecoverHandle` in the file UMacAppUtilities.incl.p rejects the handle, because `_RecoverHandle` returns a handle with a zero high byte, and `TestRecoverHandle` compares the entire handle.

   Fix: Replace the assignment:

```
TestRecoverHandle := RecoverHandle(masterPointer) = Handle(h);
```

   With:

```
TestRecoverHandle := StripLong(RecoverHandle(masterPointer)) =
                     StripLong(Handle(h));
```

6. `WithApplicationResFileDo` needs a failure handler. Since the method's **normal** behavior is to preserve the current resource file, in case of a failure it should do the same thing. The problem is that if `WithApplicationResFileDo` contains a failure handler, it must be moved to another unit; `UMacAppUtilities` cannot access `UFailure` without introducing a circular reference.

   Fix: Move `WithApplicationResFileDo` to the file UMenuSetup.incl.p and change it to the following:

```
PROCEDURE WithApplicationResFileDo(PROCEDURE DoWithResFile);

    VAR
        fi:         FailInfo;
        oldResFile: INTEGER;

    PROCEDURE HdlFailure(error: OSErr; message: LONGINT);

        BEGIN
        UseResFile(oldResFile);
        END;
```

```
BEGIN
oldResFile := CurResFile;
CatchFailures(fi, HdlFailure);
UseResFile(gApplicationRefNum);
DoWithResFile;
Success(fi);
UseResFile(oldResFile);
END;
```

7. `VisibleRect` returns the intersection of the specified rectangle along with the bounding boxes of the `visRgn` and `clipRgn`. When called during a window update, however, the `visRgn` can be smaller than expected. This difference can cause `VisibleRect` to return different sized rectangles when called inside or outside of an update event.

Fix: The final fix has not yet been determined. However, you may be able to kludge things by overriding `TWindow.Update` in the file UMacApp.TWindow.p with the following lines:

```
BEGIN
    gUpdating := TRUE;
    INHERITED Update;
    gUpdating := FALSE;
END;
```

Then, in `VisibleRect` (file UMacApp.Globals.p), change:

```
IF NOT gPrinting THEN
    SectRgn(gTempRgn, thePort^.visRgn, gTempRgn);
```

To:

```
IF NOT (gPrinting | gUpdating) THEN
    SectRgn(gTempRgn, thePort^.visRgn, gTempRgn);
```

Finally, add `gUpdating` to the file UMacApp.p, and initialize it to `FALSE` in `InitUMacApp`. Or you can just live dangerously and take out the `_SectRgn` call altogether.

8. Patching a trap with the routines in `UPatch` can cause a crash under the Finder (when MultiFinder is not present) if that trap is already patched by MacApp, because the `CleanUpMacApp` routine incorrectly restores that trap to point at the MacApp patch, rather than at the original routine.

Solution: Do not patch traps that MacApp patches (currently: `_ExitToShell`, `_InitCursor`, `_SetCursor`, `_SetCCursor`, `_GetNextEvent`, `_EventAvail`, `_StillDown`, and `_WaitMouseUp`).

Fix: Rewrite `UnpatchTrap` (file UPatch.incl.p) as follows, so it does the right thing when unpatching traps that have "newer" patches:

```
PROCEDURE UnpatchTrap(VAR thePatch: TrapPatch);

   VAR
       aPatchPtr:          TrapPatchPtr;
       newerPatchPtr:      TrapPatchPtr;

   FUNCTION GetPreviousPatchPtr(thePatchPtr: TrapPatchPtr):
                                     TrapPatchPtr;
   { Walks the patch list backwards to return the patch record
     just prior to thePatchPtr^ in the patch list }

       VAR
           tempPatchPtr:   TrapPatchPtr;

       BEGIN
       tempPatchPtr := pPatchList;
       WHILE (tempPatchPtr <> NIL) & (tempPatchPtr^.nextPatch <>
         thePatchPtr) DO
           tempPatchPtr := tempPatchPtr^.nextPatch;
       GetPreviousPatchPtr := tempPatchPtr;
       END;

   FUNCTION GetNewerPatchPtr: TrapPatchPtr;
   { returns a newer patch record in the patch list which has
     the same trapNum as thePatch }

       BEGIN
       aPatchPtr := GetPreviousPatchPtr(@thePatch);
       WHILE (aPatchPtr <> NIL) & (aPatchPtr^.trapNum <>
         thePatch.trapNum) DO
           aPatchPtr := GetPreviousPatchPtr(aPatchPtr);
       GetNewerPatchPtr := aPatchPtr;
       END;

   BEGIN
   { If this trap has a newer patch than the patch we're removing,
     then we have to take some extra special precautions. We have
     to muck with that patch's oldTrapAddr to point to this patch
     record's oldTrapAddr (for both the patch record and the
     jumpPtr code). We can pretty well ignore the case of an
     older patch on this same trap since the trap address in our
     patch record will be correct. }

   newerPatchPtr := GetNewerPatchPtr;
   IF (newerPatchPtr = NIL) THEN
       WITH thePatch DO
           NSetTrapAddress(OldTrapAddr, trapNum,
                           GetTrapType(trapNum))
   ELSE
       BEGIN
       { set up newerPatchPtr patch record so that it points to
         thePatch's OldTrapAddr }
       newerPatchPtr^.oldTrapAddr := thePatch.oldTrapAddr;
```

```
    { set up newerPatchPtr^.jmpPtr so that it jumps to where
      thePatch's code jumps to }
    IF (newerPatchPtr^.jmpPtr <> NIL) THEN
        BEGIN
        IF LongIntPtr(newerPatchPtr^.jmpPtr)^ = $2F2F0004 THEN
            T1PBlockPtr(newerPatchPtr^.jmpPtr)^.OldTrapAddr :=
                                               thePatch.oldTrapAddr
        ELSE IF IntegerPtr(newerPatchPtr^.jmpPtr)^ = $2F3C THEN
            TPBlockPtr(newerPatchPtr^.jmpPtr)^.OldTrapAddr :=
                                               thePatch.oldTrapAddr
        ELSE
            BEGIN
            {$IFC qDebug}
            Writeln('###In UnpatchTrap: can''t figure out '
                    'what kind of patch ', ORD(newerPatchPtr),
                    ' is!');
            DebugStr('Can''t unpatch trap.');
            {$ENDC}
            END;
        END;
    END;

{ Unlink the patch from the linked list of patches }
IF @thePatch = pPatchList THEN
    pPatchList := thePatch.nextPatch
ELSE
    BEGIN
    aPatchPtr := pPatchList;
    WHILE (aPatchPtr <> NIL) & (aPatchPtr^.nextPatch <>
      @thePatch) DO
        aPatchPtr := aPatchPtr^.nextPatch;
    { Couldn't find thePatch, so don't try to unpatch it. }
    IF aPatchPtr = NIL THEN
        EXIT(UnpatchTrap);
    aPatchPtr^.nextPatch := thePatch.nextPatch;
    END;

{ If the patch allocated a block in the system heap,
  deallocate it }
WITH thePatch DO
    jmpPtr := DisposeIfPtr(jmpPtr);
END;
```

9. The value for `kPriorityHigh` in the file UMacApp.p is wrong. Currently, it is initialized with the same value as `kPriorityLow`. Oops.

    Fix: Modify the file UMacApp.p so that the value of `kPriorityHigh` is set to `kPriorityNormal-32`, `kPriorityHighest+32`, or just plain `32`.

10. `IsClassIDMemberClass` does not range check for negative class IDs. This could result in some extremely rare cases where a handle appears to be an object when it really is not.

    Solution: In the file UObject.a, replace:

```
Cmp.W       (A0),D0           ; make sure class ID is in range
Bge.S       isFALSE
Cmp.W       (A0),D1           ; make sure class ID is in range
Bge.S       isFALSE
```

With:

```
Cmp.W       (A0),D0           ; make sure class ID is in range
Bge.S       isFALSE

Tst.W       D0                ; make sure class ID is non-negative
Blt.S       isFALSE

Move.W      D0,D2             ; make sure class ID is even
And         #1,D2
Tst.W       D2
Bnz.S       isFALSE

Cmp.W       (A0),D1           ; make sure class ID is in range
Bge.S       isFALSE

Tst.W       D1                ; make sure class ID is non-negative
Blt.S       isFALSE

Move.W      D1,D2             ; make sure class ID is even
And         #1,D2
Tst.W       D2
Bnz.S       isFALSE
```

11. Discipline signals a problem on a `_Get1NamedResource` call when it tries to load `CODE("GMain")`. This segment is listed in `'seg!'` and `'res!'`, but it doesn't exist.

    Fix: This bug is ultra-benign, but can be fixed by removing the reference to `GMain` in the file MacApp.r

12. The number of calls to `RegisterStdType` has increased from 17 to 25 since the MacApp 2.0ß9 release; however, the limit (`kMaxSignatures`, defined in the file UMacApp.p) remains at 32. This difference means your application can only register seven additional types instead of the 15 previously allowed.

    Fix: Re-compiling MacApp with a limit of 40 should suffice for now. Future versions of MacApp will implement a dynamic list so that no limits would be imposed.

13. In the file UMacAppUtilities.incl.p, the routine `LongerSide` always returns the height of the rectangle because of an incorrect expression.

    Fix: Replace the following line:

    ```
    IF (bottom - top) >= (left - right) THEN
    ```

    With:

    ```
    IF (bottom - top) >= (right - left) THEN
    ```

14. When compiling in non-debug mode `IsHandlePurged` is compiled as an INLINE instead of a Pascal function. Unfortunately, this INLINE does not use the same Boolean convention as the Pascal compiler. In Pascal, `TRUE = $01` and `FALSE = $00`, but the INLINE routines defines `FALSE = $00` and `TRUE = $FF`.

Fix: In UMacAppUtilities.inc1.p, replace:

```
FUNCTION IsHandlePurged(h: UNIV Handle): BOOLEAN;
    INLINE $205F,    { MOVE.L (A7)+,A0 }
           $4A90,    { TST.L (A0) }
           $57D7;    { SEQ (A7) }
```

With:

```
FUNCTION IsHandlePurged(h: UNIV Handle): BOOLEAN;
    INLINE $205F,    { MOVE.L (A7)+,A0 }
           $4A90,    { TST.L (A0) }
           $57D7,    { SEQ (A7) }
           $4417;    { NEG.B (A7)}
```

MABuild Bugs

1. MABuild does not support both AppName.r and AppName.rsrc files as part of a MacApp project. Actually, the problem is a more general one: the file Build Rules and Dependencies defines the default dependency ".rsrc ƒ .r". Therefore, if AnyFile.rsrc is mentioned either in the file Basic Definitions or your own .MAMake file, Make produces a command that compiles AnyFile.r into AnyFile.rsrc, or complains if AnyFile.r does not exist.

   Solution: Avoid the .rsrc suffix for files that are not compiled from .r files.

   Fix: Globally replace ".rsrc" with ".r.o" in the files {MATools}Basic Definitions and {MATools}Build Rules and Dependencies. This change causes Make to create Anyfile.r.o files instead of AnyFile.rsrc files, removing the conflict and preserving any .rsrc files that you may have created with ResEdit or ViewEdit. Be sure to update your .MAMake file similarly.

2. MABuild doesn't support spaces or multiple files in the `OtherViewTypesSrc` Make variable, because the following line in the file Build Rules and Dependencies:

```
IF "{OtherViewTypesSrc}" != ""
```

gets expanded to something like:

```
IF ""Whatever was in SrcApp:TSizerViewType.r"" != ""
```

The double quotes on either end cancel each other out, and any pathname with spaces is treated as separate items.

Fix: Everything should work fine if you change the default application dependency in the file Build Rules and Dependencies to:

```
"{ObjApp}{AppName}" ƒƒ ∂
    "{XAppRezSrc}" ∂
    "{BuildFlags}" ∂
    {MacAppResources} ∂
    {MacAppRIntf} ∂
    "{OtherViewTypesSrc}" ∂    # Note the quotes on this line
    {OtherRezFiles} ∂
    {OtherRsrcFiles}
```

3. MABuild doesn't support more than one user library.

   Solution: Not yet determined. Requires a change to MABuildTool.p.

4. Creating an application with `qNeedsROM128K` set to `TRUE` and running it on a 512KE under System 3.2 causes it to bomb with an ID = 12 error, because the traps that MacApp needs are not present. However, the application runs properly under System 3.4, as the traps are implemented under that system.

   Fix: Tell MacApp to use the set of glue routines that check for the presence of the needed trap before it is called. In {MAPInterfaces}UPrinting.p, replace the following lines:

   ```
   {$IFC NOT qNeedsROM128K}
   {$IFC UNDEFINED UsingPrinting} {$I Printing.p} {$ENDC}
   {$ELSEC}
   {$IFC UNDEFINED UsingPrintTraps} {$I PrintTraps.p} {$ENDC}
   {$ENDC}
   ```

   With:

   ```
   {$IFC UNDEFINED UsingPrinting} {$I Printing.p} {$ENDC}
   ```

   In {MALibraries}PrivateInterfaces:UPrinting.p, replace:

   ```
   {$IFC NOT qNeedsROM128K}
   Printing,
   {$ELSEC}
   PrintTraps,
   {$ENDC}
   ```

   With:

   ```
   Printing,
   ```

5. At the top of the file UMacAppUtilities.inc1.p are the following compiler options:

   ```
   {$W+}
   {$R-}
   {$Init-}
   {$OV-}
   {$IFC qNames}
   {$D+}
   {$ENDC}
   ```

   The intent here is that these routines should not have debugger probes (`%_BP`, `%_EP`, `%_EX`) inserted into them, allowing them to run at full speed. Unfortunately, if you compile with something like `MABuild -NoDebug -Trace`, the debugger probes are inserted.

   Fix: Add `{$D-}` before `{$IFC qNames}`

6. The Commando dialog box for MABuild is out of date. For example, `-NeedsSystem6` and `-NoDebug` are now the MABuild default and cannot be turned off through the Commando dialog box.

7. The help button in the debug options dialog box in the MABuild Commando interface is partially obscured.

8. The Commando dialog has a three-state button "Show Times", that sets the flag "-T". The help text for this is "Have all tools show elapsed time." Actually, "-T" tells only MABuildTool to show elapsed time; to have all tools do this, you need the "-TT" flag.

9. There is a small problem in the file {MAPInterfaces}UTEView.p that causes your compiles to be imperceptibly slower than you would expect. Several references to `__TEView__` at the top of the file should really be `__UTEView__`, thus:

```
{$IFC UNDEFINED __UTEView__}
{$SETC __UTEView__ := FALSE}
{$ENDC}

{$IFC NOT __UTEView__}
{$SETC __UTEView__ := TRUE}
```

10. In the file UViewCoords.h, `#ifndef __UVIEWCOORDS__` should be `#ifndef __UViewCoords__`.

    Fix: Change the header file.

11. MacApp uses CPlusLib instead of CPlusLib881 when compiling for C++ and FPU support.

    Fix: In the file Basic Definitions, replace:

```
#############
# For MPW 3.0, 3.1
#############
31CPlusSupport = ∂
     "{CLibraries}CRuntime.o" ∂
     "{CLibraries}CInterface.o" ∂
     "{CLibraries}CPlusLib.o" ∂
     "{CLibraries}StdCLib.o" ∂
     "{PLibraries}PasLib.o"

31CPlusNonFPUSANELib = ∂
     "{CLibraries}CSANELib.o" ∂
     "{PLibraries}SANElib.o" ∂
     "{CLibraries}Math.o" ∂
     "{CLibraries}Complex.o"

31CPlusFPUSANELib = ∂
     "{CLibraries}CLib881.o" ∂
     "{CLibraries}CSANELib881.o" ∂
     "{PLibraries}SANELib881.o" ∂
     "{CLibraries}Math881.o" ∂
     "{CLibraries}Complex881.o"
```

With:

```
#############
# For MPW 3.0, 3.1
#############
31CPlusSupport = ∂
    "{CLibraries}CRuntime.o" ∂
    "{CLibraries}CInterface.o" ∂
    "{CLibraries}StdCLib.o" ∂        # removed CPlusLib.o
    "{PLibraries}PasLib.o"

31CPlusNonFPUSANELib = ∂
    "{CLibraries}CPlusLib.o" ∂       # add CPlusLib.o
    "{CLibraries}CSANELib.o" ∂
    "{PLibraries}SANElib.o" ∂
    "{CLibraries}Math.o" ∂
    "{CLibraries}Complex.o"

31CPlusFPUSANELib = ∂
    "{CLibraries}CPlusLib881.o" ∂    # add CPlusLib881.o
    "{CLibraries}CLib881.o" ∂
    "{CLibraries}CSANELib881.o" ∂
    "{PLibraries}SANELib881.o" ∂
    "{CLibraries}Math881.o" ∂
    "{CLibraries}Complex881.o"
```

12. “MABuild’s mechanism for handling C++ Load/Dump is sort of lame. Why not support FPU and Load/Dump simultaneously? It’s not that hard to get working.”

    Fix: Yeah, but it used to be. So there. MABuild is trying to work around a problem that exists in CFront 3.1b3 and earlier. If you are using a later version, you can remove the safety check. Go into the file MABuildTool.p, remove the following lines, and rebuild MABuildTool.

```
{ C++ external symbol table files support }
IF fCPlusLoad & fNeedsFPU THEN
    BEGIN
    Echo('''###'' MABuild: Warning: CPlusLoad and NeedsFPU
         are incompatible.  Using NoCPlusLoad.');
    fCPlusLoad := FALSE;
    END;
```

Bugs Only In Debug Mode

These bugs occur only in debug versions of your program, and do not affect the final production version.

1. `DisposeIfHandle` fails if called with a valid, but purged, handle:

```
h := NewHandle(20);
IF h <> NIL THEN
    BEGIN
    EmptyHandle(h);
    DisposIfHandle(h); {<--PBreak: 'handle is so bad, couldn't get handle bits'}
    END;
```

Fix: In `DisposeIfHandle` (file UMacAppUtilities), add:

```
IF IsHandlePurged(aHandle) THEN { h might have been purged }
    BEGIN
    DisposHandle(aHandle);
    EXIT(DisposeIfHandle);
    END
```

Just before:

```
handleBits := GetHandleBits(aHandle);
```

This fix is not the cleanest, but it is the easiest.

2. Doctor, doctor. My application hangs if Print… is chosen while stopped in the debugger.

    Solution: Don't do that.

3. With a desk accessory open in the application heap (e.g., Option-Alarm Clock), you can enter the MacApp debugger, but it does not accept any keystrokes.

    Solution: Click in the Debug Transcript window to jumpstart it.

4. If the performance tools are on, you must turn them off with "T"oggle before "E"nding or "D"umping. Failure to do so leaves the performance tools active, although their data has been disposed.

    Solution: Always "T"oggle the performance tools off before "E"nding or "D"umping.

    Fix: Modify `PerfCmd` to turn off the performance tools when "E"nding or "D"umping.

5. `TTranscriptView` does not initialize `fFontInfo` in `CommonInit`.

    Solution: Call `InstallTextStyle` immediately after initializing.

6. `TList.GetSameItemNo` fails in debug if looking for `NIL`. With previous versions of MacApp, it was perfectly acceptable to check for a `NIL` object in a list. `GetSameItemNo` would return zero, as expected. With MacApp 2.0, there is an explicit check in debug mode that the object is valid, so passing `NIL` does not work.

    Solution: Call `GetSameItemNo` with the following wrapper:

    ```
    IF obj = NIL then
        index := 0
    ELSE
        index := GetSameItemNo(obj)
    ```

    Fix: Modify `TList.GetSameItemNo` (file TList.inc1.p) to make the same check.

7. If a failure occurs in `IApplication`, the debugger incorrectly issues the following warning:

    > "You're leaving a routine without calling Success for a handler that will be destroyed."

This message occurs because the routine MADebuggerMainEntry checks gTopHandler to see if the FailInfo record it points to is below the stack. However, this test doesn't work properly if gTopHandler is NIL, as it is in IApplication.

Fix: Add a check for (gTopHandler = NIL) in MADebuggerMainEntry (file UDebug.inc1.p). Replace the line:

```
forgotSuccess := ((which = tEnd) | (which = tExit))
                 & (StripLong(LongIntPtr(pLink)^) >= StripLong(gTopHandler));
```

With:

```
forgotSuccess := ((which = tEnd) | (which = tExit)) & (gTopHandler <> nil)
                 & (StripLong(LongIntPtr(pLink)^) >= StripLong(gTopHandler));
```

MPW 3.2 Compatibility

This section describes problems that occur when trying to build MacApp 2.0 under MPW 3.2. MacApp 2.0 was developed under MPW 3.0 and 3.1 and could not take into account changes made to MPW 3.2.

Note: Even at the time of this writing, it is unclear which of the following items will be compatibility problems. For example, item four is a problem with MPW 3.1a1, but not with MPW 3.2b1. On the other hand, item three is a problem with MPW 3.2b1, but not with MPW 3.2a1. Apple will update the status of these items with MPW 3.2 is final.

1. The file {MALibraries}PrivateInterfaces:UDebug.p needs symbol information from the file Packages.p. Under MPW 3.1, this file was automatically included when the file UDebug.p included the file Script.p in its USES statement. Under MPW 3.2, this is no longer the case, and UDebug does not compile.

   Fix: Add a reference to Packages before Script in the file UDebug.p:

```
USES
    <etc.>
    Desk, DiskInit, ToolUtils, Retrace, Memory, Resources, FixMath, Packages,
    Script, PasLibIntf, OSEvents, Traps, Perf, DisAsmLookUp, Notification;
```

2. The file UDebug.inc1.p contains the definition for the following procedure:

```
PROCEDURE JTOffProc(A5JTOffset: UNIV INTEGER;
                    VAR s: UNIV DisAsmStr80);
```

   DisAsmStr80 is declared in the file {PInterfaces}DisAsmLookup.p under MPW 3.1. It is no longer used under MPW 3.2.

   Fix: Change DisAsmStr80 to Str255.

3. In the NMRec record defined in the files Notification.c and Notification.p, nmSIcon has been changed to the infinitely clearer nmIcon.

   Fix: In UDebug.inc1.p, change the occurrence of nmSIcon to nmIcon.

4. At the bottom of the file UDebug.a, there is a line that looks like the following:

```
Case#.S          FIOINTERACTIVE,TIOFLUSH
```

`TIOFLUSH` is not supported under MPW 3.2a1, and the Assembler aborts with an error when it gets to this line.

Fix: Comment out or remove the reference to `TIOFLUSH`:

```
Case#.S          FIOINTERACTIVE ;,TIOFLUSH
```

SADE Compatibility

1. In the SADEScripts folder is a file called StepMethod. This file contains the definition of a procedure called `stepIntoMethod`, which includes the following lines:

```
break %_NEWMETHOD020.CacheOut
break %_NEWMETHOD020.TableOut
go
unbreak %_NEWMETHOD020.CacheOut
unbreak %_NEWMETHOD020.TableOut
```

MacApp 2.0 no longer defines the symbol `%_NEWMETHOD020` and SADE is not able to find it when you attempt to step into an overridden method.

Fix: Replace those lines with the following:

```
break %_NEWMETHOD.CacheOut
break %_NEWMETHOD.TableOut
go
unbreak %_NEWMETHOD.CacheOut
unbreak %_NEWMETHOD.TableOut
```

THINK Pascal Compatibility

1. In `TApplication.InModalState` and `TApplication.InModalMenuState` (file UMacApp.TApplication.p), the function result is not initialized to `FALSE` causing MacApp to think that desk accessories are in a modal state when compiled under THINK Pascal™. This problem does not occur when compiling under MPW because MPW allocates space for the function result with a `CLR.W -(A7)`; THINK Pascal 3.01 does not.

   Fix: Add an `OTHERWISE` statement setting the function result to `FALSE`, or initialize the function result to `FALSE` when you enter the procedure.

2. This isn't really a bug, but you might incorporate the following: in the file UMacAppUtilities.p, place a `{$PUSH} {$D-}` in front the `BlockSet` routine and a `{$D+}` after it. This change speeds up the execution of programs which are compiled with the MacApp debugger when running under the THINK Pascal environment. (Doing this may not be necessary if you incorporate the fix to problem #5 in the MABuild section.)

MacApp Samples Bugs

1. In the C++ version of DemoText, strings which normally appear in the About box show up in the color picker, because `kPromptStringsID` is declared differently between the Rez file and the C++ file.

2. In the file UIconEdit.inc1.p, the procedure `TIconBitMap.Free` does not call `INHERITED Free`. It should call `INHERITED Free` or the space in the heap used for the object never gets freed.

3. Instead of referring to `@fShowInvisibles`, `TTabTEView.Fields` actually refers to `@ShowInvisibles`.

Other

1. The script {MATools}CleanupDeRezzedViews misses a situation where it needs to quote a Shell variable. This problem causes the script to abort if the file you are processing contains a space in it.

   Fix: Replace the second line of the script:

   ```
   IF {1} == ""
   ```

   With:

   ```
   IF "{1}" == ""
   ```

2. Due to bug in System Software 6.0.5, attempts to perform a `_RecoverHandle` on a ROM resource fail with Virtual 2.0™ and greater than eight megabyte systems. With such a configuration, the Macintosh's ROM appears in the middle of MultiFinder's zone. This mapping confuses `_RecoverHandle`, so it attempts to do what it thinks is right by setting the current zone to be MultiFinder's zone, which is incorrect for ROM resources. Consequently, `_RecoverHandle` fails. Since MacApp calls `_RecoverHandle` on ROM resources when executing its `TestRecoverHandle` routine, MacApp applications are affected.

   This bug does not appear in earlier versions of the System. It is not yet clear whether or not it will be fixed in subsequent versions.

   Solution: Connectix is aware of the problem and may release a version of Virtual that solves it; however, this is really a System 6.0.5 problem, so the blame is on Apple.

Fix: If calls to `TestRecoverHandle` occurred only in debug mode, you could simply turn off Virtual when testing your application (or, at least, make sure you don't have more than eight megabytes of virtual memory). However, `IsObject`, which can be called even in non-debug mode, calls `IsHandle`, which calls `TestRecoverHandle`. Therefore, you may want `TestRecoverHandle` to call `_RecoverHandle` only in debug mode.

```
FUNCTION TestRecoverHandle(masterPointer: Ptr;
                           h: UNIV Handle): Boolean;

    VAR
        ...

    BEGIN
    IF qDebug & (qNeedsMC68020 | qNeedsMC68030 | (gConfiguration.processor
      <> env68000)) THEN
        BEGIN
        <Old body of TestRecoverHandle>
    ELSE
        TestRecoverHandle := TRUE;  {ƒƒƒ Can't really test this ƒƒƒ}
    END;
```

THINK Pascal is a trademark of Symantec Corporation.
Virtual 2.0 is a trademark of Connectix Corporation.

Macintosh
Technical Notes

Developer Technical Support

#281: Multiple Inheritance and HandleObjects

Written by: Larry Rosenstein

August 1990

This Technical Note answers a common question about MPW C++: "Why doesn't `HandleObject` support multiple inheritance?" It does this by giving a brief overview of how multiple inheritance is implemented in MPW C++.

What Are HandleObjects Anyway?

MPW C++ contains several extensions to "standard C++" for supporting Macintosh programming. One such extension is the built-in class `HandleObject`. An instance of any class descended from `HandleObject` is allocated as a handle in the heap. You refer to one of these instances as if it were a simple pointer; the compiler takes care of the extra dereference required because the object is really a handle.

A `HandleObject` is useful in Macintosh programming for the same reason a handle is useful. The use of handles helps prevent heap fragmentation. The nature of `HandleObject` imposes some restrictions on how you can use it in a program, however.

First, since each instance is allocated as a handle, it follows that all instances must be allocated on the heap. ("Native" C++ objects can be allocated on the stack or in the global space as well.) Consequently, you always declare variables, parameters, etc. to be pointers to the class. For example:

```
class TSample: public HandleObject {
public:
    ...
    long    fData;
};

TSample  *aSampleInstance;        // Legal
TSample  anotherSample;           // Results in a compile-time error
```

The error message the compiler generates in this case is "Can't declare a handle/pascal object: anotherSample." At first this message might seem strange, because the last two lines in this code seem to both declare objects. Actually, the first declaration is of a *pointer* to an object, not of the object itself.

The second restriction is that you must follow the usual rules for manipulating handles. In particular, you have to be careful about creating pointers to a `HandleObject` instance variable, since the object might move if the heap is compacted. If you write

```
long *x = & (aSampleInstance -> fData);
```

then x becomes invalid if the object moves. The solution in this case is to lock the object if there's a possibility of the heap being compacted. Instances of HandleObject are allocated with a call to _NewHandle, so you can use _HLock and _HUnlock to lock and unlock the object.

The third restriction is that you cannot use multiple inheritance with a HandleObject. The reason behind this restriction is not evident, however. To understand the reason, you must look at the implementation of multiple inheritance.

Implementing Multiple Inheritance

To understand how multiple inheritance is implemented, one needs a simple example. Suppose you define two classes as follows:

```
class TBaseA {
public:
    virtual void  SetVarA(long newValue);
            long  fVarA;
    ...
};

class TBaseB {
public:
    virtual void  SetVarB(long newValue);
            long  fVarB;
    ...
};
```

If you were to look at instances of these classes (see Figure 1), you would find that in each case the instance storage would contain four bytes for the C++ virtual table (vtable) and four bytes for the instance variable. Any code that accesses the instance variable (for example TBaseB::SetVarB) would do so using a fixed offset from the start of the object. (In this particular version of C++, this offset was 0; your offset may vary.)

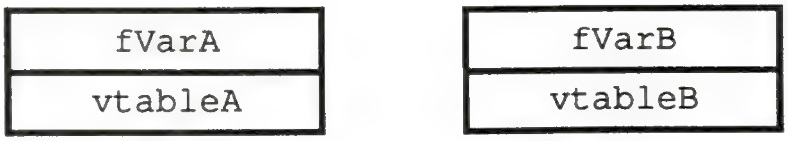


Figure 1–Layout of TBaseA and TBaseB Instances

Now suppose you define another class:

```
class TDerived: public TBaseA, public TBaseB {
public:
    virtual void  SetDerivedVar(long newValue);
            long  fDerivedVar;
    ...
};
```

In this case, an instance of `TDerived` has the following layout:

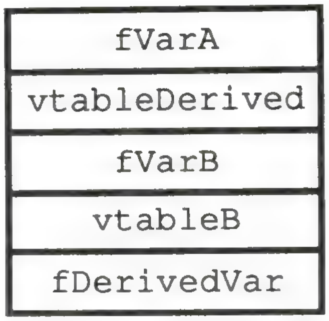


Figure 2–Layout of TDerived Instance

This is what you would expect. `TDerived` inherits from both `TBaseA` and `TBaseB`, and therefore instances of `TDerived` contain a part that is a `TBaseA` and a part that is a `TBaseB`. In addition, the virtual table `vtableDerived` includes the tables for both `TBaseA` and `TDerived`.

`TDerived` also inherits the methods defined in `TBaseA` and `TBaseB`. Suppose you wanted to call the method `SetVarB`, using a `TDerived` object. The code for `SetVarB` is expecting to be passed a pointer to a `TBaseB` object (all methods are passed a pointer to an appropriate object as an implicit parameter), and refers to `fVarB` by a fixed offset from that pointer. Therefore, to call `SetVarB` using a `TDerived` object, C++ passes a pointer to the middle of the object; specifically it passes a pointer to the part of the object that represents a `TBaseB`.

This gives you a very basic idea of how C++ implements multiple inheritance. For more details, read "Multiple Inheritance for C++" by Bjarne Stroustrup in *Proceedings EUUG Spring 1987 Conference*, Helsinki.

So What About HandleObjects?

The next question is how this implementation imposes a restriction on a `HandleObject`. The answer is simple. Each method of a `HandleObject` class expects to be passed a handle to the object, instead of a pointer. But when multiple inheritance is used, the compiler sometimes has to pass a pointer to the middle of the object. It is not possible to create a valid handle that refers to the middle of another handle. (Creating a fake handle is a compatibility risk; besides, the pointer into the middle of the handle would be invalid if the handle is moved.)

Designing a new implementation of multiple inheritance that is compatible with a `HandleObject`, as well as the rest of C++, is a big undertaking. For that reason, it is unlikely that this restriction will disappear in the future. There are, however, two alternatives to consider:

Damn the Fragmentation, Full Speed Ahead

The main reason to use a `HandleObject` is to reduce the chance of fragmentation that would result from using a non-relocatable block. In a few applications, however, the memory allocation patterns are very predictable, and fragmentation might not be an issue. In those cases, you can use "native" C++ classes. (Don't use the argument that 8 Mb machines are common, and virtual memory is here to stay so fragmentation isn't an issue at all. Data always expands to fill the available memory space, real or virtual.)

If you adopt this approach, you should read the article "Using C++ Objects in a Handle-Based World" by Andrew Shebanow in Issue 2 of *develop*, April 1990. This article describes how you can use native C++ objects and minimize heap fragmentation, by overriding the way C++ normally allocates objects. The same techniques can be used to customize the way your program allocates certain objects.

"Doctor, It Hurts When I Do That..."

The other alternative is to give up multiple inheritance. In most cases, this isn't as difficult as it sounds. The typical way you would do this is with a form of delegation. For example, you could rewrite the class TDerived as:

```
class TSingleDerived: public TBaseA {
public:
    virtual void   SetDerivedVar(long newValue);
            void   SetBaseB(long newValue);
            long   fDerivedVar;
            TBaseB fBaseBPart;
    ...
};
```

In this case TSingleDerived inherits only from TBaseA, but includes an instance of TBaseB as an instance variable. It also implements the method SetBaseB to call the method by the same name in the TBaseB class. (In effect, TSingleDerived delegates part of its implementation to TBaseB.) The advantage of this approach is that it requires only single inheritance, yet you can still reuse the implementation of TBaseB.

The disadvantages are that TSingleDerived is not a subtype of TBaseB, which means that an instance of TSingleDerived cannot be used in a situation that requires a TBaseB. Also, TSingleDerived has to define a method that corresponds to each method in TBaseB. (You can, however, define these functions as inline and non-virtual, which eliminates any run-time overhead.)

By The Way...

You should realize that the multiple inheritance implementation previously described costs some extra space, compared to a simpler implementation that does not support multiple inheritance (e.g., the implementation used for a HandleObject). Each vtable is twice as large, and each method call takes about 24 bytes, compared to 14. This is true even if you do not take advantage of multiple inheritance. For this reason, MPW C++ also contains a built in class called SingleObject, whose instances are allocated in the same way as normal C++ instance, but which only supports single inheritance. (By the way, the third class built into MPW C++, PascalObject, uses Object Pascal's run-time implementation, which takes the least amount of space, but the most execution time.)

Conclusion

You cannot use a `HandleObject` with multiple inheritance, because of the way multiple inheritance is implemented in MPW C++. Your alternatives are to give up one or the other. You can either use native C++ objects and let the objects fall where they may, or give up multiple inheritance and use a form of delegation.

Further Reference:

- MPW C++ Reference Manual
- "Using C++ Objects in a Handle-Based World," Andrew Shebanow, *develop*, Issue 2, April 1990.
- "Multiple Inheritance for C++," Bjarne Stroustrup, *Proceedings EUUG Spring 1987 Conference*, Helsinki.

Macintosh
Technical Notes

Developer Technical Support

#282: Smear Tactics

Written by: Dave Radcliffe & Carl Hewitt August 1990

This Technical Note discusses a feature of the current Macintosh hardware which will not be supported in the future. Macintosh hardware developers and driver writers should be aware of this limitation as it affects current and future products.

Current Macintosh hardware supports a feature of MC68020 and MC68030 processors which is herein referred to as “byte smearing.” Future Macintosh platforms may not support this feature, and if you have hardware or software dependent on this feature you need to revise it. Likewise you should be aware of this limitation if you are developing new hardware or software.

Spreading the Bytes Around

MC68020 and MC68030 processors have a “feature” which causes the data for byte and word transfers to be replicated (smeared) across all 32 data lines. An example illustrates the problem.

Consider the following code:

```
MOVE.L    #$12345678,D0          ; Stuff some data
MOVE.B    D0,$102                ; Write a byte of data
```

The data actually placed on the data bus, with and without byte smearing, is shown in Figure 1.

With Byte Smearing

CPU Bit	31 24	16	8	0
Byte Data	$78	$78	$78	$78
Byte Address	$100	$101	$102	$103

Without Byte Smearing

CPU Bit	31 24	16	8	0
Byte Data	xx	xx	$78	xx
Byte Address	$100	$101	$102	$103

Figure 1–Effect of Byte Smearing

With byte smearing, the byte of data is replicated across all the byte lanes; without smearing, the other bytes are undefined. A similar replication of data can occur with word transfers.

As an example where this can cause trouble, suppose you have a NuBus™ card with a device register which expects to be byte addressed at byte $102. With byte smearing it is actually possible to get away with writing a byte to any address from $100 through $103; without byte smearing, the card only sees the correct data when addressed at the correct byte $102.

Conclusion

The lack of "byte smearing" as a feature should not be a problem for most developers; after all, why would anyone write to byte $100 when they really meant to write to byte $102? Well, sad to say, at least one case of this happening has been uncovered, so if you have, either inadvertently or by design, relied on this feature, you should revise your products to run on future Macintosh platforms.

Further Reference:

- *MC68020 User's Manual,* pp. 7-9
- *MC68030 User's Manual,* pp. 7-9

NuBus is a trademark of Texas Instruments.

Macintosh
Technical Notes

Developer Technical Support

#283: A/UX System Calls From Macintosh Software

Revised by: Anathan Srinivasan & Kent Sandvik January 1991
Written by: Rob M. Smith, B. W. Hendrickson & Dave Radcliffe August 1990

This Technical Note discusses how to make A/UX system calls from applications developed in the Macintosh environment. This is useful to anyone porting an existing Macintosh driver or application to work on A/UX as well.

Changes since August 1990: Added information about how to make use of `fork()` system calls under MultiFinder, as well as how various A/UX system calls behave under the MultiFinder emulation mode.

Introduction

A/UX 2.0 now runs a broad range of Macintosh applications. The A/UX Toolbox allows most code developed for the Macintosh to run unmodified under A/UX. One exception is Macintosh device drivers. Many developers are interested in also making their Macintosh peripherals available to A/UX customers. If the peripheral requires a custom driver that accesses hardware, the driver needs to be modified to run under A/UX.

Split Decision

The A/UX Toolbox runs in "user" space in A/UX. This is a virtual, protected memory space that shares the system resources with all other processes running in "user" space. These processes are not allowed to access hardware directly. Instead, they must make a request to the A/UX kernel through a mechanism called a "system call" to deal with the hardware. The kernel, which runs in "system space," then returns data, status, etc. back to the caller. The system call is a well-defined interface that gives Unix® systems some degree of application portability.

Since any custom driver code must maintain the Macintosh interface at the Toolbox and application level, and Toolbox code cannot touch the hardware, you must split your driver into two pieces. The high-level Macintosh interface portion stays in user space, and the low-level hardware dependent, Unix-style interface becomes a Unix device driver in the kernel. So how do these two pieces communicate? They have to talk to each other through the Unix system call interface.

The code comprising the kernel portion of your driver must be adapted to do things in a "Unix way," such as providing the standard routine interface required of all Unix drivers, be multithreaded and reentrant, and not "hog" CPU time by doing "busy waits." This Note does not cover these issues, but the A/UX Device Drivers Kit (available through APDA) has example code and documentation about the topic. There are also some good books available on writing Unix drivers.

Is This A/UX or What?

If you want your code to work in either environment without change, you first need to determine if you are under A/UX at run time. The best way to do this is with the _Gestalt trap using the selector gestaltAUXVersion to determine if A/UX is the underlying operating system. Shown below is a function which returns 0 if A/UX is not present, otherwise returns the major A/UX version number (1, 2, etc.). This code relies on _Gestalt glue code available in MPW 3.2 and later.

```
/*
 *    getAUXVersion.c
 *
 *    Copyright © 1990 Apple Computer, Inc.
 *
 *    This file contains routines to test if an application is running
 *    on A/UX.  If the Gestalt trap is available, it uses that, otherwise
 *    it falls back to HWCfgFlags, which will work on all A/UX systems.
 */
#include <Types.h>
#include <GestaltEqu.h>

#define HWCfgFlags    0xB22    /* Low memory global used to check if A/UX is running */

/*
 *    getAUXVersion -- Checks for the presence of A/UX by whatever means is appropriate.
 *    Returns the major version number of A/UX (i.e. 0 if A/UX is not present, 1 for
 *    any 1.x.x version 2 for any 2.x version, etc.
 *
 *    This code should work for all past, present and future A/UX systems.
 */
short getAUXVersion ()
{
    long      auxversion;
    short     err;
    short     *flagptr;

    /*
     *    This code assumes the Gestalt glue checks for the presence of the _Gestalt
     *    trap and does something intelligent if the trap is unavailable, i.e.
     *    return unknown selector.
     */
    auxversion = 0;
    err = Gestalt (gestaltAUXVersion, &auxversion);
    /*
     *    If gestaltUnknownErr or gestaltUndefSelectorErr was returned, then either
     *    we weren't running on A/UX, or the _Gestalt trap is unavailable so use
     *    HWCfgFlags instead.
     *    All other errors are ignored (implies A/UX not present).
     */
    if (err == gestaltUnknownErr || err == gestaltUndefSelectorErr) {
        flagptr = (short *) HWCfgFlags;     /* Use HWCfgFlags */
        if (*flagptr & (1 << 9))
            auxversion = 0x100;         /* Do Have A/UX, so assume version 1.x.x */
    }
    /*
     *    Now right shift auxversion by 8 bits to get major version number
     */
    auxversion >>= 8;
    return ((short) auxversion);
}
```

A/UX Code, Under MPW?

The main system calls used to access kernel driver routines are `open()`, `close()`, `read()`, `write()`, and `ioctl()`. Of use to applications is the routine `creat()` which is included here as well. The A/UX system call mechanism is a `trap #0` with the system call selector code in register `D0`. The arguments are on the stack in the normal C calling convention, last argument pushed first.

Note that different trap calls under A/UX have different procedures concerning the use of registers and stack frames.In this Tech Note we are not trying to document each possible case, so we limit the examples to show how the registers and stack frame are usedl with the `open()`,`close()`, `read()`, `write()`, `fork()` and `ioctl()` A/UX system calls. In the case of other A/UX system calls you have to disassemble code compiled under the A/UX environment in order to find out how the parameters are passed, and how the stack frames are set.

Since MPW does not contain any A/UX libraries and doesn't know about Unix system calls, you need to use some assembly-language glue code around the trap. Following is glue code for the common A/UX routines listed above. You can extend your A/UX system call library by adding additional routines with additional system call selectors. This glue code relies on the similarity between A/UX C calling conventions and MPW C calling conventions, as well as the similarity in the sizes of parameters (`int` variables are four bytes in both systems). When these routines are entered the stack frame is already correctly set up for the `trap #0`; if you are using other languages or development systems, you may need to extend the glue to rearrange parameters on the stack to match A/UX C calling conventions.

The error code from the call is returned in `D0`. In the Unix environment, this error code is normally placed in the `errno` global variable and `D0` is set to -1 before return to the caller. Since global variables are very bad for Macintosh device drivers, this glue code relies on a special A/UX trap called `_AUXDispatch` which can return a pointer to an A/UX `errno` global variable. The C functions `SetAUXErrno()` and `GetAUXErrno()` are used to set and retrieve this value. The `_AUXDispatch` trap is defined in an A/UX include file /usr/include/mac/aux.h and you need this file to compile the C code. For more information about the AUXDispatch trap, consult the *A/UX Toolbox: Macintosh ROM Interface* manual. Lastly, all function names have been preceded by the prefix "AUX" to distinguish them from their MPW C library counterparts (e.g., the A/UX `read()` function is named `AUXRead()` here).

```
;   AUXIO.a -- Glue for A/UX I/O system calls
;
;   Copyright © 1990 Apple Computer, Inc.
;   All rights reserved.
;
;   This module contains C callable routines to execute A/UX system (trap 0)
;   calls.  The parameters to these routines is exactly as they are described
;   in the A/UX man(2) documentation.  This means all char * parameters are
;   NULL terminated C strings, not Pascal strings.  They all presume that A/UX
;   is in fact running.  Certain death will result otherwise.

            CASE    ON      ; For C
            INCLUDE 'SysEqu.a'
            IMPORT  SetAUXErrno
;
;   Here are all the routines and their C calling conventions:
;   long    AUXCreat (char *path, long mode);
            EXPORT      AUXCreat
;   long    AUXOpen (char *path, long oflag, long mode);
            EXPORT      AUXOpen
;   long    AUXClose (int fildes);
```

```
            EXPORT      AUXClose
;   long    AUXRead (long fildes, char *buf, long nbytes)
            EXPORT      AUXRead
;   long    AUXWrite (long fildes, char *buf, long nbytes)
            EXPORT      AUXWrite
;   long    AUXIoctl (long fildes, long request, long arg)
            EXPORT      AUXIoctl

;   Some local entry points
            ENTRY       auxerr
            ENTRY       auxcommon
            ENTRY       auxexit

AUXCreat    PROC
            move.l      #$8,D0          ; creat function selector
            bra.b       auxcommon       ; Join common code
AUXOpen     PROC        EXPORT
            move.l      #$5,D0          ; open function selector
            bra.b       auxcommon       ; Join common code
AUXClose    PROC        EXPORT
            move.l      #$6,D0          ; close function selector
            bra.b       auxcommon       ; Join common code
AUXRead     PROC        EXPORT
            move.l      #$3,D0          ; read function selector
            bra.b       auxcommon       ; Join common code
AUXWrite    PROC        EXPORT
            move.l      #$4,D0          ; write function selector
            bra.b       auxcommon       ; Join common code
AUXIoctl    PROC        EXPORT
            move.l      #$36,D0         ; ioctl function selector
            bra.b       auxcommon       ; Join common code

;   Trivia of the month.  The flow of the code is a little weird
;   here because of a strange interaction between the assembler
;   and the linker.  Logically, auxcommon should go here, but what
;   happens in that case is the assembler generates a byte branch
;   instruction for the previous instruction, but then the linker
;   cheerfully fills in the byte offset, which if auxcommon were
;   the next instruction would be zero.  At runtime, this causes
;   the bra.b to get interpreted as a bra.w and of course the code
;   flies off into never-never land.  So we stick in some convenient
;   intervening code to ensure the offset is never zero.
auxerr      PROC        ENTRY
            move.l      D0,-(SP)        ; Push error code
            jsr         SetAUXErrno     ; Set errno
            add.w       #$4,SP          ; Remove parameter
            move.l      #$FFFFFFFF,D0   ; Set -1 for return value
            bra.b       auxexit         ; Outta here

auxcommon   PROC        ENTRY
            trap        #$0             ; trap 0
            bcc.b       auxexit         ; CC, no error
            bra.b       auxerr          ; Do common error handling

auxexit     PROC        ENTRY
            rts
            ENDPROC

            END
```

The second argument to the `AUXIoctl` call needs some special attention. The A/UX header file /usr/include/sys/ioctl.h describes the format of `request`. These four bytes hold several fields describing the data format. Normally, macros defined in the ioctl.h header file take care of packing these fields. Make sure you use the same format when you construct your `request` argument. Just use the example commands in the /usr/include/sys/*ioctl.h files as a reference.

Following are the C functions to properly get and set the A/UX `errno` global variable:

```
/*
 *    AUXErrno.c
 *
 *    Copyright © 1990 Apple Computer, Inc.
 *    All rights reserved.
 *
 *    This file contains routines to properly get and set the standard Unix global
 *    errno from within an Macintosh application.  It uses the AUXDispatch trap
 *    to get a pointer to the address to be set.
 */
#include <aux.h>

void SetAUXErrno (err)
long err;
{
    long    *errnoptr;

    if (!getAUXVersion ())
        return;                 /* No A/UX, do nothing */

    errnoptr = 0;
    AUXDispatch (AUX_GET_ERRNO, (char *) &errnoptr);
    /*
     * If errnoptr is still NIL, AUXDispatch failed so do nothing
     */
    if (errnoptr)
        *errnoptr = err;
    return;
}

long GetAUXErrno ()
{
    long    *errnoptr;

    if (!getAUXVersion ())
        return (0);             /* No A/UX, return noerror */
    errnoptr = 0;
    AUXDispatch (AUX_GET_ERRNO, (char *) &errnoptr);
    /*
     * If errnoptr is still NIL, we're not under A/UX, or AUXDispatch failed
     * so do nothing
     */
    if (errnoptr)
        return (*errnoptr);
    else
        return (0);
}
```

Use of the fork() call under A/UX MultiFinder emulation

The following advice concerns the use of the A/UX `fork()` system call under the MultiFinder emulation mode. Under A/UX the kernel does not separate the data region of the parent process for the child after a `fork()` call. If we do a simple fork we have suddenly two MultiFinder processes running, and they both will share the same resources. The MultiFinder memory space is set up as shared memory, and since the child in UNIX inherits all shared memory segments from the parent across the fork, both the parent process and the child process will be using the same stack. This will lead to chaos if the child pushes something to the stack while the parent removes the data, or vice versa. The child should have a separate stack until we have done an `exec()`, then the child process has it's own memory world.

So what we need to do is to set up a separate data area for the child's process stack use. The child process will get its own data area by allocating enough stack space by the parent before the `fork()`, and passing this space to the `fork()` system call using a special `fork()` call, which is explained later.

The `fork()` system call copies the current stack frame of the parent onto the new stack space, resets the stack pointer to point to the new stack in the child, and then issues the trap to jump into the Unix kernel to continue to set up the new process structures. This enables the child to access information from the stack in the same manner as any other process. Details to keep in mind while using this mechanism are :

a) Allocate memory for the stack which is guaranteed not to be freed until after the child process has completed its exec.

b) Pass the address of the high memory end of the allocated memory for the stack to `fork()`, not the low memory address.

c) The address to be passed as the caller-environment argument is computed differently depending on whether the calling routine has a Pascal or a C stack frame. The examples given later show how the calculation is done.

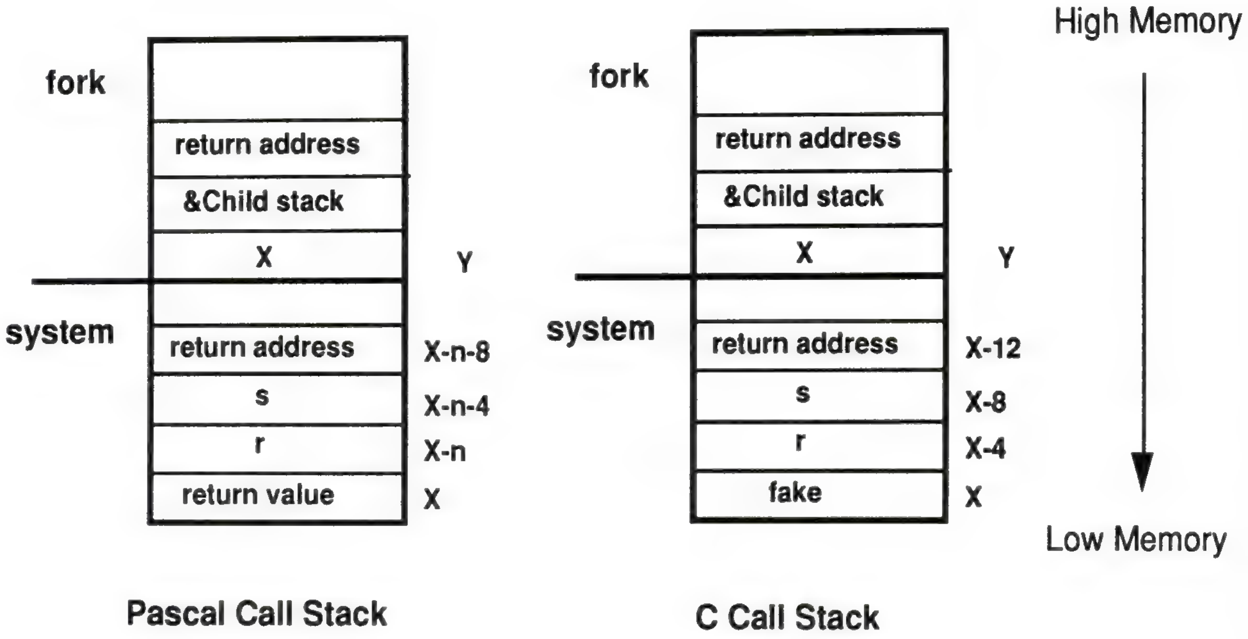
d) The calling routine needs to be very careful about what the child does before `exec()` or `exit()`. Pointers and structures accessed via the stack will point to the parent's copy, since only the local/current frame has been copied.

In particular allocation of large arrays should be done only after ensuring that the space allocated for the child stack is sufficiently large to copy the entire stack frame. This is important because arrays could be allocated on the stack, and there could exist array sizes which cause the current stack frame size to exceed that of the allocated child stack space. This will result in only part of the current stack frame being copied over onto the child. In such cases seemingly normal accesses from the child will end up being in the wrong area and cause strange behavior (the screen is locked up, bus errors are frequent etc.).

Using `malloc()` and `free()` to allocated space for such large buffers on the heap will eliminate this problem. However one needs to be aware that though the space is allocated on the heap, the space is accessed via a pointer which is on the current stack frame. This means that accesses from the child to the space in question will result in accesses to the parent's copy.

e) The parent must clean up of the allocated space for the interim stack for the child after the child has exit:ed.

The following picture illustrates how the stack parameter passing is done with a Pascal stack and a C stack:



n = sizeof(Ret. value)
X is determined thus;
X = &r + sizeof(r) + sizeof(Ret. value)

X is determined thus:
X = &fake

The design issue of returning to the caller from `fork()` (as opposed to providing a `fork()`-`exec()` combination which does not return from the fork but goes ahead and execs the required program as well) should be favored after looking into the problem carefully. Providing a separate `fork()` has advantages in the form of letting the user set up communication channels between the parent and child before `exec()`, or allowing the user to set up the appropriate environment before exec(). The problems has to do with the possibility of the not-so-wary programmer using the feature improperly and leaving two Macintosh environments running simultaneously, which will lead to chaos very quickly. Thus use of `fork()` from within an application must be done with extreme caution.

Given below is an example of the use of `AUXFork()`, a special `fork()` implementation. This example also shows how to set up the A/UX environment.

```
#define STACKBYTES 2048            /* size in bytes */
#define STACKSIZE STACKBYTES/sizeof(long)
unsigned long *childstack;

pascal long     AUXDispatch(selector,p)
short           selector;
char            *p;
extern          0xABF9;

#define AUX_GET_ENVIRON 11         /* get pointer to environ */
```

```
char **auxenviron;
extern int AUXFork(), AUXExecl(),AUXWait(), AUX_exit();

int system(s,fake)
char    *s;
int     fake;
{
        int     status, pid, w;
        register int (*istat)(), (*qstat)(), (*cstat)();
        int GetAUXErrno();
        long aux_errno;

        childstack = (unsigned long *) (NewPtr (STACKBYTES));

        /* copy the environment */
        AUXDispatch(AUX_GET_ENVIRON,(char *)&auxenviron);

        if((pid = AUXFork(&childstack[STACKSIZE],&fake)) == 0) {
                (void) AUXExecl("/bin/sh", "sh", "-c", s, 0);
                (void) AUX_exit(127);
        }
        else {
                if (pid < 0) {
                        DisposPtr((char *)childstack);      /* Fork failed */
                        return(-1);
                }
                else {
                        w = auxwait(&status);
                        DisposPtr((char *)childstack);
                        return((w == -1)? w: status);
                }
        }
}
```

In the above example, the parent sets up the space for the child stack, gets a pointer to the environment to be passed to `exec()`, and calls `AUXFork()`. A dummy variable 'fake' is passed as a parameter to `system()` to enable `AUXFork()` to copy the current stack frame on to the child stack. After the child exits, the parent cleans up the space allocated to the child stack. `AUXWait()` is used to block the parent until the child exits or terminates. The parent has to wait for the child to exit or terminate for this scheme to work properly within MultiFinder, If the child does not exit or terminate, the Macintosh environment is blocked and may lose a number of events and signals necessary to maintain its state. Thus use of fork makes sense only if we are sure that the child exits or terminates without taking too much time to execute.

The following example shows how to write `AUXFork()`:

```
;   AUXFork.a -- Glue for A/UX fork call
;
;   Copyright © 1990-91 Apple Computer, Inc.
;   All rights reserved.
;
;   This module contains C callable routines to execute A/UX fork
;   calls. This function presumes that A/UX is in fact running.
;   Certain death will result otherwise.

                             INCLUDE       'Traps.a'

                             CASE   OBJECT
```

```
                        EXPORT AUXFork

;   AUXFork routine
;
;
; pid = AUXFork(new_top_sp, caller_env)
;
;       new_top_sp:   This is one past the highest address that is
;                             in the new stack area.
;       caller_env:   This is an address on the current stack that is
;                             one past the highest address in the stack frame
;                             of the calling routine.
;
;       return values -
;        in parent:   pid == -1             failure
;                             pid == child  success
;        in child:    pid == 0
;
;
;       To call auxfork -
;               Allocate memory for the child's stack which is guaranteed not to
;       be freed until after the child process has completed its exec.  Remember
;       to pass the end of that memory region to auxfork, not the beginning.  The
;       address to be passed as the caller_env argument is computed differently
;       depending on whether the calling routine has a pascal or C stack frame.
;               Note that the calling routine needs to be very careful about what
;       the child does before exec or exit.  Only the local frame has been copied
;       and only the frame pointer has been fixed up.  For example, if the calling
;       routine has an array on the stack and uses a pointer to it for efficiency
;       then the child's pointer will point at the parent's copy, not the child's.
;       Also, if the parent must be careful not to delete or change anything the
;       child may be using.  Caveat emptor!
;
;
;       How to compute the caller_env argument -
;
;       Pascal: compute ((char*)&leftmost_argument) + sizeof(leftmost_argument)
;                       + sizeof(function return value, if any) and pass that.
;
;               e.g.    pascal Boolean system(short r, long s, long c)
;               auxfork(&new_stack[LENGTH_OF_STACK], (&r + sizeof(shor) + sizeof(Boolean)))
;
;       C:              add a fake rightmost_argument and pass the address of that.
;




;               e.g.    int system(short r, long s, long c, long fake)
;                               auxfork(&new_stack[LENGTH_OF_STACK], &fake)
;
;
AUXFork                         PROC

        ; make a copy of the stack frame
        move.l  4(a7),a0                ; just past end of new stack
        move.l  8(a7),d1                ; just past end of caller environment
        move.l  d1,d0                   ; length = end of caller
        sub.l   a7,d0                   ; ... - current stack
        sub.l   d0,a0                   ; new stack -= length of old
        move.l  a0,d0                   ; save the stack base for after copy
        move.l  a7,a1                   ; don't want interrupts to trash stack

@2      move.w  (a1)+,(a0)+             ; word aligned (it is a stack!)
        cmp.l   a1,d1                   ; done?
        bhi.s   @2                      ; ... nah, keep copying
        move.l  d0,a0                   ; ... yep, save new stack pointer


        ; now, do the fork
        move.l  2,D0
        trap    #0
        ; D1 == 0 in parent process,  D1 == 1 in child process.
        ; D0 == child pid in parent,  D0 == parent pid in child.
        bcc.b   @0                      ; did we fork?
        move.l  #-1,D0                  ; ... nah, failure
@1      rts
@0      tst.b   D1                      ; who am i now?
        beq.b   @1                      ; ... parent, get out of here

        ; ... child, so fudge registers
        move.l  a6,d1                   ; offset of fp = fp
        sub.l   a7,d1                   ; ... - old stack
        move.l  a0,a7                   ; set up new stack pointer
        move.l  a0,a6                   ; new frame pointer = sp
        add.l   d1,a6                   ; ... + offset of fp

        clr.l   (a6)                    ; the fp points to never-never land
        lea     do_exit,a1              ; and a guaranteed exit
        move.l  a1,4(a6)                ; becomes the return address

        move.l  #0,D0                   ; the child returns
        rts

do_exit move.l 1,D0
        trap    #0
        ENDP

        END
```

Issues with using A/UX system calls in the MultiFinder environment

General :

The following comments describe how various A/UX system calls behave under the MultiFinder environment:

Blocking / Sleeping system calls :

Many of the system calls can result in situations which cause the calling process to go to sleep awaiting an event which wakes it up .For instance opening a pipe from process and writing to the pipe will result in the write waiting until another process opens the pipe for reading. Such situations should be avoided when using the system calls from within a Macintosh application.

Depending on the priority at which the sleep occurs, the application can cause the entire Macintosh environment to hang (when the sleep is non interruptible), or the system call returns with error number indicating an interrupted system call. This will happen because the blocked process is sleeping at a priority from which it can be woken up by signals used to implement VBL's or other Macintosh aspects - and which is almost always bound to happen. One way to get around this problem is by using options which prevents the blocking and spin in a loop polling the result from the system call, until we are guaranteed to have a situation wherein the system call will not block. However, polling in this manner should be done only for very short intervals, and when we are sure that the polling will end in success in a short time. If this is not the case, then the application doing the polling will be stuck in the polling loop without giving up the CPU for other applications (which is extremely unfriendly MultiFinder behavior).

Caution About Blocking On Read Calls

Be aware that reads from drivers may block the calling application until some data arrives. Since the complete MultiFinder environment exists as a single process under A/UX, you do not want a pending read to block for an extended period of time. This problem is not unique to A/UX—the same thing also happens under the Macintosh OS. In a serial driver, for example, the application should check to see if any characters have been received and are waiting to be read before issuing the read call. The `read()` should then request only that many characters. This is implemented differently under A/UX than under the Macintosh OS. The available character count is determined by doing an `ioctl()` system call to the device in question. The terminal `ioctl()` commands to do this are listed in the A/UX manuals under *"termio"* in section 7. The `FIONREAD ioctl()` command returns the number of characters waiting to be read from the A/UX serial driver. This can cause problems when using the IOP-based serial driver on the Macintosh IIfx; for more information on this topic, refer to Technical Note #284, IOP-Based Serial Differences Under A/UX.

sbrk and brk:

There is no consistent way for an application to use `sbrk()` and `brk()` properly and ensure that other applications within the MultiFinder partition are aware of the new `sbrk()` and `brk()` limits and behave appropriately. Thus it doesn't make sense to use these A/UX system calls. `sbrk()` and `brk()` are mostly used to get additional data space, and this can already be achieved by using either `NewPtr()/NewHandle()` or `malloc()`.

setuid / setgid / setreuid / setregid / nice/ setgroups / setcompat / setsid / setpgid / plock/ ulimit/ phys:

These A/UX system calls have the same problem as above - i.e. we don't want to modify any process related A/UX structures/information which in turn affects all the applications running under the MultiFinder partition.

sethostid / sethostname / setdomainname / sysacct / reboot / powerdown / nfs_getfh / adjtime:

It is not recommended to affect system wide structures/data with user processes (allowed only for super user).

signal / ssig / sigvec / sigblock / sigsetmask / sigpause / sigstack / sigpending / sigcleanup :

Synchronization with signals and related calls have the same problem as earlier stated, but with additional complexities. While not providing signals would eliminate the problem of maintaining signals on a per-application basis within MultiFinder, a subset of the signals functionality has to be provided to enable applications to deal properly with certain system calls. Otherwise these calls may result in the signals being raised to indicate errors or other status information. (e.g the `SIGPIPE` signal is raised if a process sends data on a broken stream set up via the socket system call.). Signals necessary to resolve the situations mentioned earlier should be supported, but all other signals should return without accomplishing anything.

Most of the signal functionality can be accessed via the special `AUXDispatch` trap.

Pause/ alarm/ kill/ setitimer:

If only a subset of the functionality of signals is going to be provided it does not make much sense to make use of these calls.

Use of pipes :

Blocking on reading an empty pipe and blocking on writing more than `PIPE_MAX` bytes of data should not cause the Mac environment to hang (`PIPE_MAX` is defined in A/UX to 8192). These situations can be avoided in the following ways:

a) Ensure that all writes greater than `PIPE_MAX` bytes are broken up into smaller chunks (this may involve a bit of book-keeping and access to additional buffer space.).

b) Use the `fcntl()` A/UX system call to set that appropriate file descriptors returned by `pipe()` to use the `O_NDELAY` flags (or the `_NONBLOCK` semantics provided by POSIX). This guarantees that both the above cases of blocking are avoided. However, both `read()` and `write()` returns with a count of 0 which is indistinguishable from an end-of-file indication. This, along with judicious use of the polling strategy to avoid blocking mentioned above, can be used to prevent a lot of potential blocking situations.

In general use of named pipes is much simpler in a Macintosh application. This because named pipes gives the programmer the possibility to use standard Macintosh File I/O for inter-application communication. Use of regular pipes to set up communications between a parent process and related child/grandchild processes has to be done with great care. The pipe descriptors have to be set up appropriately for communication, before doing the `exec()`, but after the `fork()`. Improper usage may result in two separate MultiFinder processes running - which results in very quick deterioration of the system environment.

The requirement of cleaning up the interim child stack used during a `fork()` imposes the restriction of the parent (MultiFinder) having to wait for the child to exit. This means that all communication involving pipes between related processes must not block, and moreover must complete relatively quickly.

Messages:

Message operations should ensure that they do not cause the calling process to block. In the case that they result in blocking, the operations invariably fail and return an error number specifying an interrupted system call. The caveats mentioned about blocking hold true in situations where messages could block.

Semaphores:

Semaphores on AT&T SysV based Unix systems are fairly complicated. With the addition of further restrictions imposed by the limitations of MultiFinder running under A/UX, semaphore usage from within a Macintosh application should be attempted with utmost care. By the very nature of the operation of semaphores, sleeping/blocking situations are bound to arise. Usage of the

IPC_NOWAIT flag prevents sleeping/blocking. Thus it's possible to implement a conditional semaphore, whereby the MultiFinder process does not sleep on behalf of the application using semaphores (when it cannot do the required atomic action).

As with its usage from a regular Unix process, care should be taken to avoid situations leading to a deadlock or situations where deadlocks could happen. For instance this is true in the case where one process locks a semaphore and then exits without resetting the semaphore. Other processes will find the semaphore locked even though the process which had done the locking is no longer around. To avoid such problems the SEM_UNDO flag should be used with semaphore operations. Here again the application developer needs to be aware of the problems associated with blocking which is mentioned above.

Use of lockf:

The lockf() system call can be used if it is done judiciously. Using lockf() with the mode set to F_TLOCK is recommended; this will return with an error if a lock already exists for the region of interest to be locked.

Flock :

A request to lock (flock() system call) an object that is already locked will cause the caller to block until the lock is acquired, unless LOCK_NB (nonblocking lock) is used which results in nonblocking semantics to be applied.

Networking :

a) accept() : This call will result in the caller blocking until a connection is present if no pending connections are present on the queue, and the socket in question is not marked as non-blocking, This situation needs to be avoided.

b) recv()/recvfrom()/recvmsg() : These calls would result in the call blocking until a message arrives if no messages are available at the socket, unless the socket is marked nonblocking.

c) select() : Timeout should not be 0 - this would result in blocking indefinitely.

d) send()/sendto()/sendmsg() : These calls will block if no message is available at the socket to hold the message to be transmitted, unless the socket has been placed in the nonblocking mode.

e) socket(): Use of setsockopt() to set options on the socket connection should be done carefully. Situations which could result in the indefinite blocking should be avoided (for eg. setting SO_LINGER when the socket is opened in the reliable delivery mode would result in blocking when the socket is closed, until the socket decides that it is unable to deliver the information).

nfssvc / async_daemon:

These system calls cannot be called directly from the Macintosh world because these calls never return. To use these calls we need to first fork() a new process and then exec() a program containing this call as the child process. Additional mechanism in the form of a nonblocking wait for the parent (perhaps wait3()) needs to also be ensured.

ioctl :

The `ioctl()` A/UX system call is provided to enable programs running on Unix to access all the peculiarities of specific devices in cases where the standard I/O library lacks the necessary capabilities. Applications or programs which need to do this require device specific knowledge relevant to A/UX. The recommended way to use `ioctl()` is to write a pure Unix program, a toolbox (hybrid) program, or a small glue code snippet inside the Macintosh binary application using the `ioctl()` system call to accomplish A/UX specific functionality.

Conclusion

The routines presented here show basic techniques for accessing A/UX system services. By properly using these and other system calls, you can extend your Macintosh device drivers and applications beyond the limits of the Macintosh OS without having to ship a special version of your application for A/UX.

Further Reference:

- *A/UX Device Drivers Kit,* APDA
- *A/UX Programmer's Reference,* Section 2.
- *Writing A Unix Device Driver,* Egan & Teixeira, Wiley.
- *The Design of the UNIX Operating System*, Bach, Prentice-Hall
- Technical Note #284, IOP-Based Serial Differences Under A/UX

Unix is a registered trademark of UNIX Development Laboratories, Inc.

#284: IOP-Based Serial Differences Under A/UX

Written by: Rick Auricchio August 1990

This Technical Note discusses use of the Macintosh IIfx IOP-based serial driver under A/UX, especially under certain error conditions which cause it to perform differently than documented in `termio(7)`. (The SCC driver, used on non-IOP machines, conforms to `termio(7)` in all cases.) References to "the driver" herein refer to the IOP-based serial driver, `seriop.c`.

Bad Character Bits Are Not Passed to an Application

Because the IOP does not return the bits of a character which is received with a parity error, the A/UX driver always returns NUL to an application. Applications which use `PARMRK` mode to recover a character with a parity error always think the character was a NUL .

Break Always Returns a NUL to an Application

Because the IOP always returns a NUL character when a break occurs, the A/UX driver passes it along to an application. There is no way for the driver to determine that the NUL is superfluous. Other break processing is unaffected. Programs which use `IGNBRK` mode receive the unexpected NUL in the data stream.

Multiple Errors in a "Chunk" of Characters Are Not Reported

The IOP only reports parity and framing errors on the first such occurrence in its internal buffer. The A/UX driver always reads characters in "chunks" from the IOP; therefore, only the first error is reported and subsequent errors go unnoticed. For example, assume an application sets `PARMRK` mode, expects to read a 10-byte packet (call it "ABCDEFGHIJ"), and four parity errors occur during transmission of the packet. (The received data is "ABxDExxHxJ," where the x characters are parity errors). The IOP returns six valid data characters, marking only the first bad character ("ABxDEHJ"). The subsequent errors go unreported, simply causing missing characters. The A/UX driver then marks the bad character (marking it as NUL as previously described), and returns "ABmmxDEHJ" (where mm are the 0xff 0x00 marker bytes) to the application.

This situation causes two problems:

- Since only nine bytes, and not 10, are returned to the application, it is possible for the application's read to block permanently. This would occur if the application simply issued a 10-byte read request.
- The application has no way of knowing that several characters were dropped.

Since the timing of the A/UX driver's "chunk reads" and the arrival of data can vary, there is no way to predict or prevent the occurrence of this problem.

The Current "DMA Hang" IOP Code Patch is Incomplete

The existing IOP code has been patched, both under the Macintosh OS and A/UX, to code around a bug in its DMA logic. Should any errant characters be received **during** servicing of a DMA operation, the IOP silently discards them, never reporting any error. This can cause unreported dropping of characters. Since this situation is timing-dependent, there is no way to predict or prevent the occurrence of this problem.

IIfx Serial Switch cdev

The IIfx Serial Switch cdev does not, itself, work under A/UX. If users need to enable "Compatibility" mode, they should do so first under the Macintosh OS. A/UX, upon booting, honors the switch setting in parameter RAM. Refer to Technical Note #271, Macintosh IIfx: The Inside Story for more details on the IIfx Serial Switch cdev.

Further Reference:

- *A/UX System Administrator's Reference*
- Technical Note #271, Macintosh IIfx: The Inside Story

Macintosh
Technical Notes

Developer Technical Support

#285: Coping With VM and Memory Mappings

Revised by: Craig Prouse April 1991
Written by: Craig Prouse February 1991

The purpose of this Note is twofold. First, it describes in detail how to use the `GetPhysical` routine. This routine is critical to the support of alternate bus masters on certain machines without Virtual Memory (VM) and all machines with VM. Included is an ancillary discussion of several closely-related VM routines. Second, it reiterates a number of issues important to VM compatibility and elucidates some of the deeper VM issues of which specialized developers should be aware. Compatibility issues are especially important for developers of SCSI drivers, NuBus™ master hardware, and code which runs at interrupt time.

Changes since February 1991: This update incorporates new issues which have come up during System 7.0 beta testing, and it also attempts to clarify some issues which have proven to be particularly troublesome or widely misunderstood.

Everybody Must Get Physical

If you are developing NuBus expansion cards with bus mastership or direct memory access (DMA) capabilities, and if you have ever done development or compatibility testing with Apple's recent machines, like the Macintosh IIci and Macintosh IIsi, you have undoubtedly noticed some strange behavior. You might tell the card to dump data into a buffer at $00300000 and the data instead appears at $006B0000. "What's happening here?" you must ask yourself.

Well, there's a new game in town—it's called a **discontiguous physical address space**. What that means in simple terms is that there is potentially a big hole in memory. If you have eight megabytes installed in a Macintosh IIci, for instance, that memory appears to the CPU and to NuBus in two separate 4 MB ranges: [$00000000–$003FFFFF] and [$04000000 – $043FFFFF]. Everything from the end of Bank A to the beginning of Bank B is essentially empty. Bank B memory does not start until at least $04000000.

To compensate for this, the operating system uses the memory management unit (MMU) to map all the **physical** memory (what the hardware sees) into a single contiguous **logical** address space (what all Macintosh code sees). The logical address space looks exactly like the memory map you've known for years. The translation is completely transparent to software. If you're an applications developer and you read the low-memory global at $10C, you don't care that the address that the processor actually looks at is $0400010C. When the processor originally put a value in that spot, it went through the same translation. Everything is relative and you always get just what you'd expect.

The sole exception is for software which runs on the Macintosh but communicates addresses to NuBus master hardware. Say, for instance, that you have developed a video frame grabber which dumps an image into a handle you've allocated for that purpose. When you call `_NewHandle` with an argument of `frameSize`, you get back a logical address. If you use a 68030, or a 68020 with a 68851 PMMU, to store data into that handle, the MMU performs an address translation and places data into a corresponding physical address. NuBus hardware, however, does not use the MMU's address mapping tables. If your driver passes along a logical address from the Memory Manager, the frame grabber does not know to translate it (indeed it cannot), and the logical address is interpreted as a physical address. External hardware may dump a beautiful captured image well outside your carefully allocated handle and perhaps right across the top of MacsBug and other similarly important things. Bugs like this are extremely difficult to isolate unless you understand their behavior and anticipate them.

The point is, now you must be sure to always convert logical addresses to corresponding physical addresses before passing them to any alternate bus master. A new function to support this is `GetPhysical`, which is documented in *Inside Macintosh*, Volume VI. "Great," you say, once you've read the documentation. "But `GetPhysical` is a System Software 7.0 feature and a Virtual Memory feature to boot. What do I do for System 6.0.x or if I'm not running VM?"

I'm glad you asked, because VM and the memory architecture of the Macintosh IIci are related topics. `GetPhysical`, a routine required by the IIci, is one of a suite of functions dispatched by a trap called `_MemoryDispatch` ($A05C), which is the same trap used by the major VM calls. Because some machines require `GetPhysical` even without VM, those machines have a limited form of `_MemoryDispatch` implemented in ROM.

You **can** call `GetPhysical` under System 6.0.x or under System 7.0 even when VM is not running—all you must do first is check to see that the `_MemoryDispatch` trap is implemented. If this trap is implemented, it is there for a reason, and you should use it. Although `GetPhysical` is present only for certain machines without VM, it is present and required for all machines running VM. If you update your code to be compatible with the IIci and IIsi in the 6.0.x world, you are already doing part of what is required to be compatible with Virtual Memory and System Software 7.0.

Holding and Locking Memory Versus Locking Handles

Virtual Memory introduces two new concepts—holding and locking a range of virtual memory. These are not to be confused with locking a handle. Locking a handle prevents the handle from changing its logical address during Memory Manager operations. Holding and locking virtual memory affects how VM deals with arbitrary ranges of memory during paging operations.

Holding and locking memory (as opposed to a handle) are VM functions exclusively and are accomplished with four new `_MemoryDispatch` routines: `HoldMemory` and `LockMemory`, and the corresponding routines to undo these operations, `UnholdMemory` and `UnlockMemory`. Pay special attention any time you hold or lock a range of memory that you subsequently unhold or unlock the same range. Every single call to `HoldMemory` or `LockMemory` must be balanced by a corresponding `UnholdMemory` or `UnlockMemory` because the operating system supports multiple levels of locking and holding, much like it supports multiple levels of cursor obscuration with `_ShowCursor` and `_HideCursor`.

Holding a range of memory guarantees that the data in that range is actually somewhere in physical Macintosh RAM and that no paging activity is necessary to load it. This is critical for tasks which run at interrupt time, since paging activity should not be initiated at interrupt time. VM is not guaranteed to be reentrant, and because interrupts may occur in the middle of paging, any data accessed by an interrupt handler should reside in a held block of memory. Only hold memory which legitimately needs to be held though, because any memory which is held becomes ineligible for paging. This reduces the space VM has to work with and may significantly impact system performance. Some interrupt-time tasks are deferred by VM until paging is safe, so memory they touch does not always have to be held. These tasks are called out below, in the section "Compatibility With Other Device Drivers and Interrupt-Level Code."

Locking a range of memory is more severe than holding it. This not only forces the range to be held resident in physical RAM, but also prevents its logical address from moving with respect to its physical address. This is important for drivers which initiate DMA transactions, because there must be a known, static relationship between logical and physical addresses for the duration of such an operation. Part of the behavior of `LockMemory` is to make the associated memory non-cachable which is important for DMA transfers.

Warning: Apple cannot make the point too strongly that memory should only be held or locked when absolutely necessary, and only as long as necessary. It is worth restating that the impact on performance can be significant or even fatal in severe cases. It is a crime against the machine to hold or lock memory unnecessarily. Failure to unhold or unlock memory previously held or locked is most heinous.

In non-VM environments, there is no page swapping activity. This is similar to all of memory being locked, except that caching is still enabled. Truly locked memory is neither cached nor paged. If you are running System Software 7.0 with VM, you **must** explicitly lock a range of memory with `LockMemory` before calling `GetPhysical`. You may only call `GetPhysical` on a locked block of virtual memory, or you get an error, since, among other reasons, any paging activity could invalidate the results of a `GetPhysical` call. Although it is not necessary to call `LockMemory` before `GetPhysical` if VM is not running, `LockMemory` may still be used for its favorable effect of disabling caching. This Note includes a code template (located at the end) which illustrates a "way rad" method to implement driver calls to a generic NuBus master card. It doesn't even have to know if VM is running. Hardware and drivers should be designed to support this method for maximum VM friendliness.

There is one more VM routine of interest, `LockMemoryContiguous`, which is provided to assist developers whose DMA hardware is not capable of transferring blocks of arbitrary size or for some other reason cannot use a generalized algorithm such as the one provided. Apple can only warn developers that `LockMemoryContiguous` is potentially an expensive operation in terms of performance and is one very likely to fail since contiguous physical memory may be difficult, if not impossible, to find. `LockMemoryContiguous` is not particularly useful, unless VM is running, should a range of memory happen to cross a physical discontinuity like that found on a Macintosh IIci. No hardware or software product should require VM in order to run. `LockMemoryContiguous` might be useful for determining whether a range of logical memory is actually physically contiguous, although `GetPhysical` can do the same thing without actually locking the memory.

Apple's primary recommendation regarding `LockMemoryContiguous` is to avoid its use if at all possible. If you must use `LockMemoryContiguous`, Apple recommends that you allocate your buffer as early as possible (preferably at startup) and lock it down contiguously at that time. VM is an entropic system, meaning its pages tend to become shuffled over time, so it's easiest to find contiguous memory early in a session.

When to Call?

`HoldMemory`

- Before taking control of the SCSI bus.
- Before accessing memory at interrupt time.
- To keep **critical** ranges of memory resident for performance reasons.

LockMemory

- Rarely. (Always UnlockMemory as soon as possible.)
- Before calling GetPhysical.
- Before initiating a DMA transfer.

LockMemoryContiguous

- Never, if you can help it. (If necessary, do so as early as possible—see text above).

When Not to Call?

HoldMemory

- To keep **large** ranges of memory resident for performance reasons.

LockMemory

- Before dereferencing a handle. (LockMemory should not be confused with _HLock.)
- When you really mean HoldMemory.

What Form Of Address To Pass?

All _MemoryDispatch routines described above work as expected in either 24-bit mode or 32-bit mode. In 24-bit mode, for instance, master pointer flags or other garbage bits in the high-order eight bits are ignored and taken to be zero. When switching between 24-bit and 32-bit modes, remember to use _StripAddress as outlined in Technical Note #213, _StripAddress: The Untold Story.

Special Considerations

The GetPhysical call in ROM and system software currently supports only logical RAM. This excludes the ROM, I/O, and NuBus spaces from the set of addresses GetPhysical knows how to translate. Unfortunately, machines like the Macintosh IIci and Macintosh IIsi use the MMU to map a small amount of physical memory into NuBus space so that it looks like a regular video card. Ideally one might like to use GetPhysical to get the actual RAM address of the video buffer (to provide DMA support for certain multimedia products and graphics accelerators), but the current ROM implementation of GetPhysical returns a paramErr (-50) in response to logical NuBus addresses.

Because its ROM is derived from that of the Macintosh IIci, the Macintosh LC may appear to have _MemoryDispatch implemented. This doesn't make sense, however, because the LC has no MMU. Although System Software 7.0 patches _MemoryDispatch in this case to make it unimplemented, PrimaryInit code and SCSI drivers which run before system patches are installed could be affected. Code running at this time should qualify the existence of _MemoryDispatch with the existence of an MMU, using _Gestalt.

In order to solve both of these problems as cleanly as possible, the MPW libraries contain an enhanced version of GetPhysical with greater flexibility than the ROM version.* Although the

* At this writing, the enhanced GetPhysical code has not yet been incorporated into beta versions of the System 7.0 interface libraries. This code will be made available at the earliest opportunity and this Note will be revised to indicate its availability. If you need GetPhysical to operate on RAM-based video buffers or you need to call

enhanced version is the same as the ROM version in most cases, it provides extra validation checks to guarantee stability before system patches are installed, and it applies alternate mechanisms to determine the physical address of a RAM-based video buffer. You should therefore call `GetPhysical` where it is indicated, even for address spaces where the ROM version is known to return an error. The glue code may pick up the slack or a future ROM might not return an error. In any case, your code should always be prepared to cope with any of the `GetPhysical` error results documented in *Inside Macintosh*. Remember always to call `LockMemory` before calling `GetPhysical`, and `UnlockMemory` as soon as possible afterwards.

VM Compatibility

Compatibility With Accelerator Upgrades

The burden of compatibility has long been on the shoulders of accelerator manufacturers. VM may present some additional compatibility challenges for these manufacturers.

Virtual Memory requires services which are not present in the ROMs of 68000-based machines, so VM is not supported by the Macintosh SE, even one with a 68030 accelerator. The same is true of the Macintosh Plus, the Macintosh Classic, and the Macintosh Portable. There is no guarantee that these older machines will ever be able to support VM. For practical reasons, Apple has chosen not to implement VM in a wholly ROM-independent manner. In the foreseeable future, only machines in which Apple intended to include memory management units can support Virtual Memory. Machines never intended to include an MMU do not have all the ROM code required by VM.

Virtual Memory depends on low-memory globals to indicate the presence of a memory management unit at a very early stage of the boot process. In some cases, the low-memory globals are not properly set by the boot code in ROM if the hardware features of an accelerator are significantly different from those of the stock Macintosh. The most likely problems are exhibited by 68000 Macintoshes, 68020 Macintoshes with 68030 accelerators, and Macintoshes with 68040 accelerators. There is third-party virtual memory software which provides much of the VM functionality of System Software 7.0, and which is also compatible with accelerator products. In some cases this software may be bundled with the accelerator.

Apple is not saying that VM does not work with any accelerator, but rather that the System 7.0 implementation of Virtual Memory in general does not support accelerators. Some accelerator products may work or may be modified to work. Apple simply does not guarantee that any particular accelerator product works with VM.

Compatibility With Removable Media

Obviously it would be a disaster if a user ejected the cartridge containing his backing store (paged out memory) and handed it to a coworker to take home. This would be much worse than giving away a floppy, to be faced with the "Please insert the disk…" alert. Someone would actually have part of the computer's memory in his briefcase—try to type Command-period and get out of that one. To guard against this possibility, ejectable media are not permitted to host the VM backing store. Users of removable cartridge drives are not wholly excluded, however. The driver software for such a drive may impose software interlocks to prevent ejection and indicate in the drive queue that the cartridge is nonejectable. VM accepts any sufficiently large, block oriented device as long as it is not ejectable.

`GetPhysical` as part of a PrimaryInit or SCSI driver initialization, you should be certain to take defensive measures against the special cases described above.

Compatibility With SCSI Code

Virtual Memory introduces new requirements for some SCSI hard disk drivers. Users of Apple hard disks may need to update their drivers with a System Software 7.0–compatible Apple HD SC Setup application. Third-party hard disk drivers may also need to be updated. It is up to these third parties to determine what enhancements, if any, are required for their drivers and to provide updates to their customers if necessary.

For SCSI disk driver developers, one requirement for VM compatibility may be summarized as follows (special thanks to Andy Gong for the detailed analysis):

> On System 6.0.x and earlier, all calls to the SCSI disk driver came from the file system. This being true, and the file system being single-threaded, only one SCSI disk driver would be called at any one time. Virtual Memory changes this scenario because it makes calls to the driver directly, avoiding the file system. This implies the possibility of SCSI drivers being reentered.
>
> For a SCSI driver to function correctly in the VM environment, the driver must have complete driver data separation at least on a drive-by-drive basis. Such separation makes the driver reentrant on a drive-by-drive basis. If the driver supports multiple HFS partitions on the same physical drive, the driver must be completely reentrant if any of the HFS partitions are to be used for the VM backing file.

All this means is that a driver which controls multiple drives or partitions must maintain separate driver variables to reference each drive or partition. Otherwise, the state of a transaction to one drive may be lost when the driver is reentered to service another drive. There is no problem with reentrancy for drivers which control only a single drive or partition.

In many cases of SCSI code incompatibility, reentrancy is not the problem. This affects only the small number of SCSI disk drivers which are designed to control multiple drives or partitions from a single driver. A more common problem is caused by a page fault while the SCSI bus is busy. Since VM depends on the SCSI bus to handle a page fault, a page fault is forbidden to happen while the SCSI bus is busy. Code which uses the SCSI Manager needs in general to ensure that all its code, buffers, and data structures (including TIBs) are held in real memory before taking control of the bus.

In the normal course of events, the system heap is held in real memory. Other critical structures are held for you automatically, like any range of memory passed to a Device Manager `_Read` or `_Write` call in `ioBuffer` and `ioReqCount`. So if your SCSI code is written as a device driver, and the buffer's address and length are passed in the normal driver fashion, and if your driver code and data structures are located in the system heap, you should be fully VM-compatible already (as long as you only operate on one drive per driver).

If your SCSI code is not a standard Device Manager driver or if you reference buffers as `csParams` to `_Control` or `_Status` calls, you'll need to do some extra work. Also, Apple does not guarantee that the system heap will always be held for ever and ever, so if you come to revise your driver you should seriously consider holding explicitly everything you touch while you own the SCSI bus and everything you might knowingly touch at interrupt time; and of course you should correspondingly unhold all these structures upon releasing the bus. Be a good citizen.

In addition to the requirement for reentrancy across drives served by a single driver, the driver for a disk used as a backing store must load at the earliest possible opportunity. Drivers which defer installation until INIT time are too late to be used by VM.

Compatibility With Other Device Drivers and Interrupt-Level Code

The primary concern for device drivers is that they commonly run at interrupt time and it is absolutely essential that interrupt-level code does not cause a page fault. To avoid this, drivers should make certain that any data structures they keep or reference at interrupt time are held in physical memory as described earlier. Locking the structures is typically not necessary except in cases where alternate bus master hardware accesses those structures as well.

To improve performance and compatibility with existing software and drivers, the first release of System Software 7.0 always holds the entire system heap in physical memory. No special measures need be taken if your driver and its associated data structures are all installed in the system heap. If your driver uses memory statically allocated above `BufPtr`, it may need to explicitly hold the appropriate ranges of memory to avoid paging at interrupt time. Please be aware that future versions of the Macintosh System Software may **not** hold all of the system heap automatically and it is a good habit to hold explicitly memory you know you access at interrupt time.

The Device Manager deals with `_Read` and `_Write` calls for you, and ensures that the buffers specified for such calls are safe. However, if a buffer is passed as a `csParam` to `_Status` or `_Control` calls, the Device Manager cannot do anything about it. Buffers referenced this way must be held explicitly if they are to be accessed by interrupt-level code.

Certain code types are always deferred until times when paging is safe, and as such don't have to be concerned about whether memory they touch is guaranteed to be held. Those code types include Device Manager I/O completion routines, Time Manager tasks, VBL tasks, and slot VBL tasks. The trade-off is in real-time performance. Clearly, since these tasks may be deferred, there is an increased possibility of latency which may be unacceptable for some pseudo–real-time applications. (The Macintosh has never supported true real-time processing.) An arbitrary function which might cause a page fault at interrupt time can be deferred explicitly by calling it via the trap `_DeferUserFn`.

The `_DeferUserFn` trap is asynchronous in nature, so subsequent code may be executed before the deferred function completes. If the results of a deferred function are vital to the code which follows, the deferred function needs to signal the calling code when it completes.

Apple Desktop Bus I/O requests are deferred until a time when paging is safe unless VM is certain that all code and associated data structures are located in the system heap. This is required because the ADB Manager normally processes incoming data at interrupt time and there is a potential for page faults if the service routine code or other data structures are not held in real memory. The only problem with this strategy is reduced performance for specialized ADB drivers which require most of the ADB bandwidth and don't live in the system heap. Nonetheless, it's worth mentioning.

One final note of interest pertains to a longstanding anomaly in the Device Manager. As it turns out, when you make an asynchronous `_Open` or `_Close` call to a device driver, any completion routine you supply is never called. Since Virtual Memory patches `_Open` and `_Close`, and generates an entry for the completion routine in the user function queue, the implication is that the user functions are never executed and the queue may simply fill up. There is little reason to call `_Open` or `_Close` asynchronously with a completion routine (it never would have amounted to anything anyway), so the workaround is simple: don't do it.

Compatibility With the BufPtr Method of Static Allocation

Inside Macintosh, Volume IV describes, on page 257, a method of static allocation for resident drivers or other data structures. This method has been very popular with a number of developers. The main thing for developers to remember about this method in conjunction with VM is that memory allocated in this way is not held in physical memory by default. It must be explicitly held, unlike memory in the system heap which the operating system automatically holds, at least in the first release of System Software 7.0.

When allocating memory above `BufPtr`, always use the equation defined in *Inside Macintosh*. The actual configuration of memory at boot time is much more complicated than the illustration indicates, especially with System Software 7.0 and VM. The System 7.0 boot code passes a specially-conditioned version of `MemTop` to system extensions, which guarantees that the equation has valid results. For this reason, do not use `MemTop` to determine the actual memory size of the machine; use `_Gestalt` instead. You may use `MemTop` to determine RAM size only if `_Gestalt` is not implemented, and then only at INIT time. (Apple continues to point out that good application software should not need to know this information except under extremely rare circumstances.)

Due to the way memory is organized with VM in 24-bit addressing, you may not be able to achieve nearly as much memory above `BufPtr` as you would think possible for a given virtual memory size. This is due to the possibility of VM fragmentation, which is discussed later. Without VM, the available space above `BufPtr` is generally somewhat less than half the amount of memory installed in the machine. With 24-bit VM, the available space may be significantly less, and is probably far less than one half of the virtual memory size. The "conditioning" of the `MemTop` variable takes this into account.

Compatibility With 32-Bit Addressing

To make the most valuable use of Virtual Memory, 32-bit addressing is extremely important. Needless to say, it is critical that all developers test their applications, drivers, and all other types of code extensively under System 7.0 while running 32-bit addressing—both with and without VM. Four megabit SIMMs are becoming less and less expensive, and the day is not far off when machines with at least 16 MB will be common. Correct behavior with 32-bit addressing is critical to the acceptance of both System 7.0 and developer applications. It is not acceptable to ask users to reboot with 24-bit addressing in order to use your hardware or software. For a few classes of applications it may be necessary to turn VM off in order to run **efficiently**, but VM should not prevent an application from running at all. Be sure to include a `'SIZE'` resource in your application. It should proclaim your 32-bit compatibility to the world, not to mention the Finder.

User Tips and Helpful Hints for Living With VM

Apple suggests that Virtual Memory runs more efficiently with at least four megabytes of physical RAM. Although System Software 7.0 runs on two-megabyte systems, using VM on such a system may result in unacceptable paging performance and hard disk thrashing. After holding the system heap and other RAM which must remain resident, there is simply not enough room left for efficient paging. Fortunately, with the recommended four or five megabytes, most users should be able to run arbitrarily large virtual memory environments, with little or no annoyance from paging delays and limited primarily by the sacrifice in disk space.

Virtual Memory trades virtual RAM size for some degree of performance. VM users should be aware that VM is not always a viable alternative to physical RAM. For example, an application which makes heavy use of an entire 8 MB partition for image processing may execute very sluggishly on a machine with only 4 MB of real RAM. (The benefit of VM in this case that such an application runs at all on a machine with limited RAM.) On the other hand, the same machine may concurrently run six or seven different megabyte-plus applications with little or no appreciable performance degradation except when switching among them. (This is where VM really shines.) Performance is determined by virtual RAM size versus physical RAM size with the memory access dynamics of each application thrown in as a wild card. Each VM user will find a combination of settings which he or she finds most comfortable.

A Special Note Regarding 24-Bit VM

Some machines in the installed base are capable of running VM, but do not have 32-bit clean ROMs and must run with 24-bit addressing. What this means to users who want to run VM is that they can only take advantage of 14 MB of virtual memory. That's all there is room for in a 24-bit address map. More likely the limit is 12 or 13 MB because every installed NuBus card eliminates 1 MB of virtual RAM address space. (The way VM increases RAM size with 24-bit addressing is—more or less—by making each unused NuBus slot look like a 1 MB RAM card and making ROM and each installed NuBus card look like a nonrelocatable 1 MB application partition.)

You can be a real friend to the Process Manager (formerly known as MultiFinder) by taking care in which slots you install NuBus expansion cards: ROM always occupies one megabyte at $800000, limiting the largest contiguous block of virtual memory to somewhat less than eight megabytes. The balance may be in a contiguous block as large as four or five megabytes unless it is fragmented by a poor selection of slots for expansion cards. Best results are achieved by placing all expansion cards in consecutive slots at either end of the bus—this has the effect of collecting all the immovable one megabyte rocks into a single pile where one is less likely to trip over them. Haphazard placement of NuBus cards may generate a number of one or two megabyte islands interspersed throughout the upper portion of the virtual memory space, and that does **not** help to run more applications or to manipulate larger objects.

In machines with fewer than six NuBus slots, recall that one "end" of the bus is actually in the middle of the slot address space. In a Macintosh IIcx, slots are numbered $9 through $B. Expansion cards should be installed from the lowest-numbered slot up (contiguous with the ROM) to avoid fragmentation. In a Macintosh IIci, slots are numbered $C through $E. This poses a greater problem. Due to the RAM-based video in virtual slot $B, it is nearly impossible to avoid some degree of fragmentation when using the built-in video option. When not using this option, installing NuBus cards from the highest-numbered slot down (at the end of memory) is the best course. Fortunately, the IIci ROM supports 32-bit addressing. In 32-bit addressing VM, none of this discussion applies. Virtual Memory and NuBus do not share space in the 32-bit address map.

A Template for GetPhysical Usage

A great deal of the justification for this code may be inferred from the code itself and the comments within. The basic rules are all covered in the previous text, but the simmered-down algorithm *sans* error handling is this:

```
See if there is _MemoryDispatch;
If there is _MemoryDispatch:
    LockMemory the interesting range of memory;
    If the memory is locked:
        Loop:
            Call GetPhysical on memory;
            Loop:
                Process a physical block;
            Until all physical blocks have been processed;
        Until all memory is translated;
        UnlockMemory the interesting range of memory;
Otherwise:
    Process the block of memory the way you used to;
End.
```

```
PROGRAM GetPhysicalUsage;

  USES Types,Traps,Memory,
    Utilities;    { see DTS sample code for TrapAvailable }
    {In beta versions of the 7.0 interfaces, also use VMCalls, now in Memory.}

  CONST
    kTestHandleSize = $100000;

  VAR
    aHandle     : Handle;
    aPtr        : Ptr;
    aHandleSize: LongInt;
    hasGetPhysical: Boolean;
    lockOK      : Boolean;
    vmErr       : OSErr;
    table       : LogicalToPhysicalTable;
    physicalEntryCount: LongInt;
    index       : Integer;

  PROCEDURE SendDMACmd(addr: Ptr; count: LongInt);

    BEGIN
      { this is where you would probably make a driver call to }
      { initiate DMA from a NuBus master or similar hardware   }
    END;

  BEGIN
    aHandle := NewHandle(kTestHandleSize);
    IF aHandle <> NIL THEN BEGIN
      MoveHHi(aHandle);
      HLock(aHandle);
      aPtr := aHandle^;
      aHandleSize := GetHandleSize(aHandle);

      hasGetPhysical := TrapAvailable(_MemoryDispatch);
      { if GetPhysical is available it should always be used  }
      { without it, DMA fails on IIci and many later machines }
      IF hasGetPhysical THEN BEGIN

        { must lock range before calling GetPhysical }
        { Call LockMemoryContiguous instead of LockMemory if a single physical  }
        { block is required, but beware! This is inefficient and failure-prone! }
        vmErr := LockMemory {Contiguous} (aPtr,aHandleSize);
        lockOK := (vmErr = noErr);
        IF NOT lockOK THEN BEGIN
          { handle LockMemory error indicated by vmErr }
        END;

        IF lockOK THEN BEGIN
          table.logical.address := aPtr;
          table.logical.count := aHandleSize;
          vmErr := noErr;
          WHILE (vmErr = noErr) & (table.logical.count <> 0) DO BEGIN
            physicalEntryCount := SizeOf(table) DIV SizeOf(MemoryBlock) - 1;
            { this makes it easier to change "table" to include more    }
```

```
     { MemoryBlocks -- defaultPhysicalEntryCount is a suggestion }

     vmErr := GetPhysical(table,physicalEntryCount);
     { GetPhysical returns in physicalEntryCount the number   }
     { of physical entries actually used in the address table }

     IF vmErr = noErr THEN BEGIN
      FOR index := 0 TO (physicalEntryCount - 1) DO
       WITH table DO
        SendDMACmd(physical[index].address,physical[index].count);
     END
     ELSE BEGIN
      { handle GetPhysical error indicated by vmErr }
      { loop will terminate unless vmErr is negated }
     END;
    END;

    { always unlock any range you lock! }
    IF Boolean (UnlockMemory(aPtr,aHandleSize)) THEN;  { ignore UnlockMemory err }
   END;

  END
  ELSE
   { no GetPhysical, life is bliss }
   { remember how easy this used to be before GetPhysical? }
   SendDMACmd(aPtr,aHandleSize);
 END;

END.
```

Further Reference:

- *Inside Macintosh*, Volume II, Memory Manager
- *Inside Macintosh*, Volume IV, Initialization Resources
- *Inside Macintosh*, Volume VI, Compatibility Guidelines
- *Inside Macintosh*, Volume VI, Memory Management
- Technical Note #213, _StripAddress: The Untold Story
- Technical Note #261, Cache As Cache Can

NuBus is a trademark of Texas Instruments
THINK is a trademark of Symantec Corporation

Macintosh
Technical Notes

Developer Technical Support

#286: The Serial General-Purpose Input (GPi)

Written by: Craig Prouse February 1991

This Technical Note discusses the latest supported methods for reading, validating, and configuring the GPi serial input across all members of the Macintosh family.

GPi is a software-configurable serial input present on some machines. It is located at pin 7 on the DIN-8 serial connectors, and connects to the DCD input of the Z8530 Serial Communications Controller (SCC). Because DCD is monopolized by the mouse on the Macintosh Plus, GPi is not implemented on that machine. Other machines which do not support GPi include the Macintosh Classic and Macintosh LC. On these machines, pins 7 of the DIN-8 serial connectors are not connected.

Reading GPi (The Easy Part)

A number of developers currently make use of the GPi input on the serial ports of the Macintosh SE, Macintosh II, and Portable families. It's a handy feature and DTS regularly receives the question of how to read this input. The code required is actually quite simple, assuming all the proper hardware support is in place. As stated previously, some Macintosh models do not support GPi. For those machines which do support GPi and for which the SCC chip is directly accessible, the following code reads the state of GPi.

```
        movea    (SCCRd).w,a0   ; best place to get address of SCC RR0
        move.b   aCtl(a0),d0    ; modem port--use bCtl for printer port
        btst     #3,d0          ; GPi comes in DCD input--bit 3 of SCC RR0
        beq      @GPi0
GPi1    ...
        ...
GPi0    ...
```

This is currently the only way to determine the state of the GPi serial input. There is no support for this signal in the Serial Driver. If the SCC is not directly accessible, then neither is GPi. To determine if the SCC is accessible, check with _Gestalt. If an SCC exists but is not accessible, _Gestalt claims that there is no SCC.

Validating and Configuring GPi (A Little Bit Harder)

To aid application developers in determining whether a machine supports GPi, a _Gestalt selector is available in System 6.0.7 and later. This selector is fully documented in *Inside Macintosh*, Volume VI, and specifies (a) whether GPi is supported on port A, (b) whether GPi is supported on port B, and (c) whether GPi may be used as a clock input for synchronous modems on port A.

There is another new call which developers can use to configure GPiA as an external clock. Previously, developers had to manipulate a bit in VIA1 to enable or disable external clocking on this pin. Unfortunately, there has always been some ambiguity about the sense of this bit (the SE uses the opposite sense of the Macintosh II) and the VIA bit is not present at all on the Macintosh IIfx—see Technical Note #271, Macintosh IIfx: The Inside Story. The friendly way to configure GPiA uses _HwPriv selector 7, as documented in that Technical Note.

MPW has never defined a high-level calling interface to this particular trap macro, and no glue has ever been available for Pascal and C programmers. Until this is remedied, the following inline glue fills in quite nicely:

```
FUNCTION SwapSerialClock (clock, portID: Integer) : Integer;
  INLINE $205F, $7007, $A198, $6B02, $3008, $3E80;

pascal short SwapSerialClock (short clock, short portID) =
{
  0x205F, 0x7007, 0xA198, 0x6B02, 0x3008, 0x3E80
}
```

For the normal 3.672 MHz internal serial clock, pass $0000 in the clock parameter. For external clocking provided at the GPiA pin, pass $0001 in the clock parameter. Other clock sources are theoretically possible, so use only one of these two values.

Only one value is currently supported for the portID parameter, and that is the Serial Driver enumerated constant sPortA. If necessary, this constant must be casted to type short or coerced to type Integer, according to the terminology of your development language.

If an error results, SwapSerialClock returns a negative number, otherwise it returns the previous GPiA configuration which is a non-negative number. This makes it convenient to save and restore the original state.

SwapSerialClock works with system software back to 6.0.5, although it does not achieve the desired results on the Macintosh IIfx. In fact, it may crash. This is a problem which is addressed in System Software 7.0. All the features described in this Note are technically new features for System 7.0, but Apple encourages developers to employ them if necessary (and available) in 6.0.x-compatible applications and suggest to their customers to use the latest available system software to obtain maximum benefit from these types of applications.

The following code fragment shows how to use these new features without explicitly depending upon specific system software versions. It assumes only that the _Gestalt trap is implemented or emulated by MPW glue (which is already available). It is not necessarily possible to trap the error of calling SwapSerialClock on a Macintosh IIfx with pre-7.0 software. It is best to avoid executing this code at all on such a configuration or else risk a system crash.

```
PROGRAM SerialClock;

  USES Types,GestaltEqu,Serial;

  CONST
    internalClock = 0;      { convenient constants for SwapSerialClock }
    externalClock = 1;

  VAR
    gestErr     : OSErr;
    hasGPiAClk : Boolean;
    oldClockMode: Integer;
    result     : LongInt;

  FUNCTION SwapSerialClock(clock,portID: Integer): Integer;
    INLINE $205F,$7007,$A198,$6B02,$3008,$3E80;
  { this could be supported in a future version of MPW }
```

```
BEGIN
  gestErr := Gestalt(gestaltSerialAttr,result);
  IF gestErr = noErr THEN BEGIN
    hasGPiAClk := (band(result,bsl(1,gestaltHasGPIaToDCDa)) <> 0);
    IF hasGPiAClk THEN BEGIN
      { SwapSerialClock is supported if gestaltHasGPIaToDCDa is supported }
      { it may experience difficulties with Mac IIfx and pre-7.0 systems… }
      oldClockMode := SwapSerialClock(internalClock,Integer(sPortA));
      IF oldClockMode < 0 THEN BEGIN
        { handle case of error setting the clock mode }
      END;
    END
    ELSE BEGIN
      { handle case where there is no GPiA clock support }
    END;
  END
  ELSE BEGIN
    { handle case where Gestalt doesn't know about serial attributes }
    { this usually means assume no support, or ask for later system… }
  END;
END.
```

Further Reference:

- *Inside Macintosh*, Volume III, The Macintosh Hardware
- *Inside Macintosh*, Volume VI, Compatibility Guidelines
- *Guide to the Macintosh Family Hardware*, Serial I/O Ports
- Technical Note #129, _Gestalt & _SysEnvirons—A Never Ending Story
- Technical Note #271, Macintosh IIfx: The Inside Story
- Technical Manual: *Z8530 SCC Serial Communications Controller* (contact Zilog or AMD)

Macintosh
Technical Notes

Developer Technical Support

#287: Hey Buddy, Can You Spare A Block?

Written by: Philip D. L. Koch, Jim Reekes, & Kenny Tung

February 1991

This Technical Note discusses a new feature of the System Software 7.0 Disk Initialization Package—bad block sparing.

Warning: Software that accesses blocks directly from the disk or makes assumptions about the physical blocks of a device is, has always been, and will always be, a compatibility risk. The format of the file directory is changing in System Software 7.0 and additional changes being made to the Disk Initialization Package will cause such software to fail.

Introduction

The Disk Initialization Package that is being shipped with System Software 7.0 contains a new feature, **bad block sparing**. This new feature is done without modification to any disk drivers and is independent of the device's geometry. When the system finds bad blocks, it removes them from the free storage pool so that the file system does not use them. This feature does not affect any applications which use the normal HFS file system; the only expected impact of this feature is for those applications which perform disk reads from or writes to the disk directly, like scavenger, recovery, and floppy disk utilities.

The new feature of the new Disk Initialization Package maps any bad blocks found on any HFS volume. This feature is valuable considering the number of blocks that are on a 800K or 1440K floppy disk. If a single block is bad, the previous Disk Initialization Package would return an error and the system would reject the entire disk. The new Disk Initialization Package may attempt to map any bad block found on a non-Sony drive, but the probability is low that a newly formatted SCSI drive has a bad block. If it were to have bad block remaining after its low-level format, then there's something wrong with the disk and it's most likely a hardware problem. It is possible for a volume to have encountered a bad block during normal use. These can be mapped out by going to the Finder and choosing Erase Disk… from the Special menu, calling `_DIBadMount`, or calling `_DIZero`. Calling `_DIFormat` or `_DIVerify` does not call upon the sparing algorithm.

Note: When a user chooses Erase Disk… from the Special menu, the Finder calls `_DIBadMount` with the error code in the `evtMessage` set to `noErr`. This causes the sequence of Disk Initialize dialogs to appear. Applications can call `_DIBadMount` to reformat a disk and ensure that bad blocks are spared. This is further documented in the Disk Initialization Package chapter of *Inside Macintosh*, Volume VI.

The sequence of events after calling `_DIBadMount` is as follows. `_DIBadMount` issues a call to the disk driver to perform the low-level format of the disk. This is a `_Control` call with `csCode = formatCC`. Once the driver returns its result, `_DIBadMount` attempts to verify the blocks on the disk. This becomes a _Control call to the driver with csCode = verifyCC. If the driver returns an error, then _DIBadMount begins scanning the disk for bad blocks and mapping them out. Afterwards, a directory is created and the volume information is written to the disk. This completes the process of _DIBadMount.

The Disk Initialization Package does not perform the low-level formatting that is required by a SCSI device or any other non-Sony drive. Generally SCSI drivers ignore these csCode values mentioned above and return noErr. In returning noErr, _DIBadMount performs no bad block sparing. Thus, it isn't likely that sparing is used by _DIBadMount on a non-Sony disk. Such sparing is performed by the utility software supplied with the disk. Issuing the SCSI Format command causes the drive controller to perform the low-level format and sparing of any bad blocks in hardware. This is handled by the SCSI formatter software included with the hard disk product. If any bad blocks are suspected on a SCSI device, it is recommended that users format the disk with the supplied SCSI utility. Finally, **good** SCSI drivers map bad blocks dynamically so that any bad block encountered during normal use is removed.

If an application calls _DIFormat directly, the Disk Initialization Packages performs no bad block sparing. If the disk does contain bad blocks, the disk cannot be used. _DIVerify is used to verify if the disk contains any bad blocks. This disk may be used if these bad blocks are mapped out. To do this, the application has to use either _DIBadMount or _DIZero.

The Algorithm

Disks that are error-free are initialized exactly as before. Only when the driver's verify routine fails during _DIBadMount or if _DIZero encounters bad blocks is the sparing algorithm invoked. Sparing proceeds by making a second pass over the disk, writing and then reading back a test pattern. Testing is done a single track at a time, as a compromise between speed and wasted space. (Since it is impossible to determine the geometry of a SCSI drive, all disks larger than the Floppy Disk High Density (FDHD) are tested at an assumed track size equal to the FDHD.) If there are any errors or retries during a test, the sectors are deemed bad.

If more than 25% of the disk is found to contain bad blocks, if the I/O errors appear to be due to hardware failure rather than media failure, or if certain critical sectors are bad (see below), then the initialization fails as it would have without sparing. Otherwise, the blank HFS volume structure is written to the disk. Because the sectors touched during this operation are included in the "critical" list, no changes to the code which initializes the logical volume structure are required. After the volume structure has been written, the Disk Initialization Package:

1. Removes the bad spots from the volume bitmap of available free storage.

2. Creates file extent descriptors for the bad spots and inserts them into the volume extent B*-tree so that the free-space scavenging that takes place at volume mount and by Disk First Aid do not attempt to reintroduce the bad spots into the free storage pool. A reserved file ID (5) is used for these extents.

3. Sets a flag (hexadecimal 0200) in the HFS volume header attributes to provide a canonic and simple way for applications to determine whether or not the disk has been spared. The bad extents are described in the B*-tree with reserved file ID=5.

4. For 800K floppies only, the number of allocation blocks is reduced by one (from 1594 to 1593). This change is done to prevent previous Finders from doing disk-to-disk copies physically (sector-by-sector), which would fail trying to copy the bad blocks. The Finder does physical copies only on 1594 block disks as an optimized method of disk copying.

The critical sectors (those that must be good even on a spared disk) include the boot blocks, the HFS master directory and spare master directory, the volume bitmap, and the initial extents for the two HFS B*-trees. In practice, this means that the first 50 sectors and the second to the last sector of a 1440K FDHD disk must be good.

As described later in this Note, the most error-prone region of a floppy disk seems to be the inner tracks on side one (the bottom side), which, unfortunately, is where HFS keeps the alternate copy of the Master Directory Block (MDB). The normal sparing logic rejects an entire track if any sector on it is bad, but to improve the algorithm's effectiveness, the algorithm has a special case for the alternate MDB. Even if there is an error somewhere in the alternate MDB's track, the algorithm does not reject the disk if the sector it is on is useable. This means that with *n* sectors per track, only about 1/*n* of the disks with damaged inner tracks are rejected by the sparing algorithm. On FDHD disks *n*=18, so about 95% of the damaged disks can be initialized, assuming only one sector on the track is bad.

Experience Has Shown

In the few weeks after the new Disk Initialization Package was written, Apple had the opportunity to test it against roughly one hundred disks (both 800K and 1440K) that could not be formatted with the standard Disk Initialization Package, or that had developed errors since being formatted. Most of these were successfully spared using the new Disk Initialization Package.

Engineers noticed a trend in the small sample tested: errors on both 800K and 1440K disks are not uniformly distributed across the media. By far the most common place for errors to occur is on side one tracks 75-79, with the second most common place being tracks 0-3. Interior errors seem to be rare. One explanation for the high proportion of errors on side one tracks 75-79 might be that Apple now parks the heads over that area before ejecting a disk, so when a disk is inserted the heads come down there, possibly touching and scraping the media. Side one is probably more vulnerable both because the diameter of a given track is smaller on that side and because the head faces up and therefore easily collects dust.

The Compatibility Risks

Applications that manipulate disks through the HFS documented routines (by far the vast majority of applications) see no difference using spared disks. The only user-visible difference is that the Disk Initialization dialog box shows an additional message ("Re-Verifying disk…") while sparing an imperfect disk, and a spared disk has slightly less free space than an error-free disk. The identified risks associated with sparing are due to the following:

1. Applications that directly manipulate the HFS volume structure (in particular the extent B*-tree, volume bitmap, or the volume attributes field) need to be changed to respect the bad spot extents. In particular, this includes disk utilities such as Disk First Aid, that repair and reorganize HFS volumes. Note that these same applications also need to be revised to correctly handle System Software 7.0 aliases. Apple has a version of Disk First Aid that supports both sparing and aliases, and it ships as part of System Software 7.0.

2. Applications that physically access disks, sector by sector, do not work if the disk contains a sector that has been spared. The best, and possibly only, examples of this class of application are the disk duplicating utilities.

Conclusion

Unfortunately, the first and last tracks (the ones most likely to be bad) are also the very tracks that are critical to the HFS layout and which, therefore, must be error-free. As a result, the sparing algorithm is not as effective as it might be. To minimize this problem in the future, Apple intends to change the parking space to which the Sony drivers move the heads before ejecting a disk, this time from cylinder 79 to cylinder 40. Cylinder 40 is probably a slightly better choice, as the media is more flexible at this point since it is farther from the rigid metal hub, yet it is far enough away from the outer edge to avoid the problems experiences with cylinder 0. More importantly, though, cylinder 40 does not cover any sectors that are critical to HFS.

The decision not to create a directory entry for the bad spot file has both advantages and disadvantages. The major advantage is that logical operations such as directory enumerations and file-by-file disk copies are completely unaware of the bad spots. Since Finder disk copies are file-by-file, this is important. The major disadvantage is that Disk First Aid and other third-party disk utilities do need to be upgraded to recognize and avoid the bad spots, even though they are not part of a file. The required changes, which are very simple, have been incorporated into the System Software 7.0 version of Disk First Aid.

Another, completely independent way to deal with imperfect media is at the driver and controller level. If the driver reserved a few cylinders for revectoring bad tracks, it could present the appearance of error-free media to its clients, including HFS and the Disk Initialization Package. Most, if not all, hard disk drives already do this. This is a very attractive solution, because no changes to higher-level software are required since the traditional Macintosh model of error-free media is preserved. However, it is a much more difficult task to modify the several existing floppy disk drivers (such as the 6502-based IIfx driver) than it was to enhance the Disk Initialization Package. Also, there are major compatibility problems involved with driver-level solutions since old drivers would not be able to correctly handle revectored tracks. Fortunately, this is not an mutually-exclusive decision: the disk initialization algorithm described here does not prevent later improvements to the drivers.

It should be noted that sparing in Disk Initialization Package does not address problems that occur after a disk has been formatted. This class of dynamic errors is much harder to deal with, since neither the Macintosh OS file system nor its clients are set up to handle such I/O errors gracefully, if at all. On the other hand, many operating systems apparently spare only during initial formatting. Media errors encountered during an I/O operation are handled by the driver code.

Further Reference:

- *Inside Macintosh*, Volume II, Disk Initialization Package
- MPW SonyEqu.a interface files

Macintosh
Technical Notes

Developer Technical Support

#288: NuBus Block Transfer Mode sResource Entries

Written by: Guillermo Ortiz

February 1991

This Technical Note describes the `sResource` entries needed in a declaration ROM to inform NuBus™ masters when a board is capable of receiving or sending block transfers.

Introduction

In addition to normal long word transfers, the NuBus specification defines a number of block transfer transactions. In block mode transfers, the system arbitrates for the bus a single time and then performs a group of consecutive long word transfers before releasing the bus. The reduction in bus arbitration time can result in considerable gains in performance.

Currently, Macintoshes do not support block transfers to or from NuBus cards; however, in the future, this might change. In addition, present NuBus cards can act as bus masters and initiate card-to-card block transfers (e.g., 8•24 GC Card to 8•24 Display Card). The problem is that the master needs to determine what block transfer capabilities a slave has (and future systems may want to ascertain the same). This Note describes the mechanism that is to be used for NuBus cards to register their block transfer capability.

This Note uses video boards as an example, but hardware developers should note that the same principle applies to other types of NuBus boards (e.g., memory expansion, data acquisition, etc.). Apple recommends reviewing the NuBus specification to clarify details about master transfers, locked transfers, and block transfer sizes.

Give Or Take?

There are two long word `sResource` entries which define the block transfer capabilities of the board or mode. The first describes **general block transfer information** and the second describes the **maximum number of transactions for locked transfers** (if the board supports them). If the entries specifying block transfer information are omitted, the master should assume that the target board does not support block transfers and should not test for this capability when the entries are not present. It is highly encouraged that new boards being developed do include this information since future system software will most probably only use these entries to decide if a board supports block transfers or not since any method of directly testing the board to identify its capability is liable to cause data loss or weird behavior, including system crashes.

The second word is not necessary if the board or mode does not support locked transfers.

The NuBus specifications establish that when a slave board that does not support block transfers receives such a request, it should terminate the first transfer with `/ACK`; boards that do not support block transfers and do not implement an early `/ACK` block termination must have the `sResource` block transfer information present with all the slave transfer size bits set to zero.

The format of the general block transfer information is a long word whose structure is as follows:

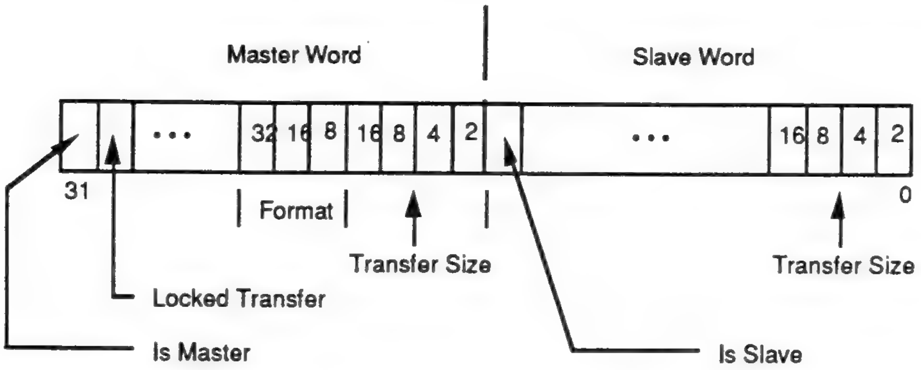


Figure 1–General Block Transfer Information Long Word Format

The fields have the following meaning:

Field	Meaning
Is Master	1 if board can initiate transactions (ORing of Master Transfer Size bits)
Is Slave	1 if board can accept transactions (ORing of Slave Transfer Size bits)
Transfer Size	Each bit indicates the number of long words per block transfer; bit set to 1 if the size is supported
Locked Transfer	1 if board can initiate locked transfers
Format	Reserved

Table 1–Descriptions Of General Block Information Fields

The Maximum Locked Transfer Count is a long word.

Figure 2–Maximum Number Of Transactions Long Word Format

How Do You Define Them; Where Do They Go?

The block transfer capability long words are kept in a card's declaration ROM. You can use `OSLstEntry` (OffSet List Entry) macros to describe both block transfer capability long words. The macro takes two arguments: the ID byte and a label designating the destination and uses them to create a long word entry. The macro puts the first argument, the ID, as is, into the high byte, and, with the second argument, calculates the 24-bit signed offset value to the destination label, putting it into the next three bytes.

If the card can support **all** block transfers in **all** of the operation modes that it supports, the block transfer capability entries are kept in one centralized place—the board `sResource` list. For example, this is the way it is done on the Apple 8•24 GC Display Card. When the board `sResource` is used to store the entries, use these ID values for the general block transfer information and maximum locked transfer count long words:

```
sBlockTransferInfo          = #20 = $14
sMaxLockedTransferCount     = #21 = $15
```

The following code fragment illustrates a board `sResource` case implementation:

```
_sRsrc_Board
    OSLstEntry     sRsrc_Type,_BoardType
    OSLstEntry     sRsrc_Name,_BoardName
    OSLstEntry     sBlockTransferInfo,_BTInfo
    OSLstEntry     sMaxBlockTransferCount,_BTMaxCnt
    DatLstEntry    BoardId,BoardId
    OSLstEntry     PrimaryInit,_sPInitRec
    OSLstEntry     VendorInfo,_VendorInfo
    OSLstEntry     SecondaryInit,_sSInitRec
    OSLstEntry     sRsrcVidNames,_sVidNameDir
    DatLstEntry    EndOfList,0
    . . .
_BTInfo
    DC.L           allBlockTransfers
_BTMaxCnt
    DC.L           maxLockedTransferCount
```

where, for example, `allBlockTransfers` = $C00F800F and `maxLockedTransferCount` = maximum transaction size. It is important to note that this value depends on the capabilities of the board under consideration as indicated in the illustrations.

If the card only supports block transfer in some modes (specifically, screen depths in the case of video boards), the information is placed in the `sResource` entries corresponding to those modes (e.g., video `sResource` parameter lists) that support block transfers. This is the way it is done on the Apple 8•24 Display Card, since it does not support block transfers in the 24-bpp mode or any convoluted interlaced mode.

The Apple `sResource` ID numbers for this case are:

```
mBlockTransferInfo          = #5 = $5
mMaxLockedTransferCount     = #6 = $6
```

The following code fragment illustrates one video parameter list within one `sResource`:

```
_EZ4u
    OSLstEntry     mVidParams,_Parms
    DatLstEntry    mPageCnt,Pages
    DatLstEntry    mDevType,ClutType
    OSLstEntry     mBlockTransferInfo,_BTInfo
    DatLstEntry    EndOfList,0
    . . .
_BTInfo
    DC.L   allSlaveBlockSizes
```

where `allSlaveBlockSizes` = $0000800F. Note that the maximum block transfer count does not need to be specified for slave devices, and for this reason it is not used in the example.

Conclusion

Cards that support block transfers must use these `sResource` entries in their declaration ROMs to allow other NuBus boards to utilize this capability thus improving compatibility and performance.

Further Reference:

- *Designing Cards and Drivers for the Macintosh Family*, Second Edition
- *IEEE Standard for a Simple 32-Bit Backplane Bus: NuBus*

NuBus is a trademark of Texas Instruments

Macintosh
Technical Notes

Developer Technical Support

#289: Deaccelerated _CopyBits & 8•24 GC QuickDraw

Written by: Guillermo A. Ortiz

January 1991

This Technical Note discusses conditions that may cause `_CopyBits` to slow down when QuickDraw acceleration is on via the Apple 8•24 GC Display Card.

Introduction

When a drawing call is issued, GC IPC (Interprocess Communication) takes control of the call and passes it to GC QuickDraw. After the normal port set up (which involves caching the port parameters if this is the first drawing call after the port was set), GC QuickDraw returns control to the application through the IPC and performs, in parallel, the drawing to its own monitor as well as any other monitors that may be affected by the operation. The application then continues its execution, probably issuing more drawing calls that get executed in the same asynchronous manner.

The result of this mode of operation is improved performance, since the application gets control back immediately after issuing the call and the GC QuickDraw moves video data to its own video buffer as well as that of other cards in a more rapid manner by using block transfers and without requiring any action by the main processor.

_CopyBits Conforms To The Same Scheme, Except...

`_CopyBits` conforms to the same operational scheme, but there are some instances in which GC QuickDraw cannot perform the call in parallel; in this cases it is even possible to suffer a performance loss, since the whole call may have to be completed before control is given back to the application and GC QuickDraw has to make calls and access data across the NuBus™.

The situations that compromise GC QuickDraw parallel operation are as follows:

- When the destination device has a `SearchProc` installed and the source color environment is different from the destination environment.

  QuickDraw calls a `SearchProc` whenever the source and destination have different depths and when two indexed pixel maps have different color tables, even though their depths may be identical. When GC acceleration is enabled, these conditions cause the following two types of behavior, dependent upon the source pixel map:

- If the source is an indexed pixel map, then GC QuickDraw executes the part of the setup that involves calling the `SearchProc`, returns control to the main processor, then completes the call in parallel. The act of calling the `SearchProc` before returning control makes the call slower than when no `SearchProc` is involved, since parallel operation does not occur throughout the whole call.

- If the source is a direct RGB pixel map, then GC QuickDraw has to call the `SearchProc` for every pixel that is drawn, and the application does not regain control until after the call to `_CopyBits` has been completed.

- When the source or destination is offscreen and not created using a `GWorld`.

  GC QuickDraw has no way to detect when an application is going to manipulate a pixel map it has created in memory, so if it has to draw to or copy from such a `PixMap`, GC QuickDraw has to complete the operation before returning control to the application.

  This behavior is contrary to the case when using a `GWorld` for offscreen environments, since in the case of a `GWorld`, GC QuickDraw is alerted by the call to `_GetPixBaseAddr` that the application is getting ready to directly change the pixels. This is the reason why it is so important that applications call `_GetPixBaseAddr` **every** time they are about to manipulate a `GWorld` pixel map directly.

- When the source `PixMap` has a color table that uses indexes that refer to a palette.

  QuickDraw now allows a color table to have indexes that point to entries in the palette associated with the destination window; when bit 14 in the `ctFlags` field is set, the value fields in the color table are treated as palette entries. When such a `PixMap` is the source for `_CopyBits`, then GC QuickDraw has to make a number of calls to the Palette Manager as part of the setup before returning control and completing the call.

  This case is similar to that of a indexed pixel map when a `SearchProc` is involved; therefore, it only implies a partial loss of parallelism; it is good to keep in mind that this case can only occur when the source `PixMap` is indexed.

Further Reference:

- *Inside Macintosh*, Volumes V & VI, Color QuickDraw
- *develop*, "Macintosh Display Card 8•24 GC: The Naked Truth," July 1990.
- Technical Note #275, 32-Bit QuickDraw: Version 1.2 Features
- Developer Notes for the Macintosh Display Cards 4•8, 8•24 and 8•24 GC (APDA, M085TLL/A)

NuBus is a trademark of Texas Instruments.

#290: Custom WDEF and wDraw

Written by: Vincent Lo

February 1991

This Technical Note explains why custom window definition functions may not respond to a `wDraw` message from the system (if you follow the documentation in *Inside Macintosh*).

Problem & Solution:

Inside Macintosh, Volume I-299, documents the declaration of the window definition function (WDEF) as follows:

```
    FUNCTION MyWindow(varCode: INTEGER; theWindow: WindowPtr; message:
                      INTEGER; param: LONGINT) : LONGINT;
```

On the first examination of the parameters, one may assume `param` always contains a long value; however, when the system is calling the WDEF with `message = wDraw`, it only stores a short value in `param` without clearing the high-order word of `param`. If the high-order word contains any value other than zero, the content of `param` is different from what the WDEF expects.

For the custom WDEF to work correctly, it should use only the low-order word of `param` when `message = wDraw`.

Note: This problem exists in all systems up to and including System Software 7.0. The suggested fix is valid for all of these systems.

#291: CMOS On Macintosh LC PDS

Written by: Paul Baker & Rich Collyer

February 1991

This Technical Note provides PDS card developers with some important information about making PDS cards for the Macintosh LC.

Due to the way the Macintosh LC was designed, Apple strongly recommends that all PDS cards be developed with CMOS parts.

Why?

Apple recommends using CMOS parts because of the way the timing works inside the V8 ASIC. When a CPU access to VIA1 takes place at exactly the same time as a VRAM transfer cycle, the LS245 buffers between the memory bus and the processor bus are disabled three clocks before the end of the CPU cycle. Since all the parts on the bus are CMOS, even though the bus is not driven, it does not return wrong data because bus capacitance keeps the bus in the correct state.

If an expansion card is added to the system, and that expansion card uses a TTL buffer, the load caused by the TTL part can cause the bus to discharge, resulting in wrong read data. If expansion cards are built using CMOS buffers or CMOS ASICs, then these problems are avoided and all are happy.

Solution

Therefore, all expansion cards for the Macintosh LC should have CMOS drivers on the high byte of the data bus. This can be done either by using a CMOS buffer, such as a 74ACT245, or by driving the bus directly from a CMOS gate array. Since this only shows up when the CPU is accessing VIA1 at the same time as a video refresh happens, it does not show up frequently. For this reason, it would be a good idea to test any system with a new I/O card by writing a tight loop that reads VIA1 and verifies correct data over a several second period. This would ensure that the data bus is not being discharged by the expansion card.

#292: Bus Error Handlers

Revised by: Colleen K. Delgadillo — May 1992
Written by: Wayne Meretsky and Rich Collyer — February 1991

This Technical Note discusses bus errors and how applications and drivers should deal with them.

Changes since February 1991: Discussion of why declaration ROMs are necessary in NuBus™ design. This discussion is important for those who are considering using a workaround instead of declaration ROMs. Also added are some hints that you should be aware of if you are planning to write a bus error handler for the '040.

Introduction

Bus errors occur within Macintosh systems under a variety of circumstances: virtual memory page faults, NuBus transfer acknowledgment other than complete (error, time-out, try-again-later), SCSI blind transfer handshaking, and so on. At present, Apple has not documented a single model for handling bus errors, and as a result, system software, applications, and device drivers use various techniques that do not always work together and therefore compromise system integrity. The following is the second revision of a definitive statement on bus errors and bus error handling.

What Is a Bus Error?

A bus error is an event that forces termination of a bus cycle. In some cases the processor takes no programmer-visible actions, as may happen if a prefetching bus cycle results in a bus error but the prefetched data is not executed, or as may happen during a cache fill burst. Typically, the termination of a bus cycle by a bus error does result in the processor taking programmer-visible actions. These actions, collectively called Bus Error Exception Processing, include termination of the instruction that caused the bus error, creation of an exception stack frame on the appropriate system stack, and transfer of control to the bus error handler designated by the current bus error vector. Bus Error Exception Processing does **not** include the execution of the bus error handler.

The cause of bus errors is different on different members of the M68000 family. On all members up to and including the MC68020, the only cause of bus errors is assertion of the `/BERR` signal (note that the assertion of `/BERR` and `/HALT` indicates a bus retry operation, not a bus error). On the MC68030, bus errors can be caused by either the assertion of the `/BERR` signal or by the internal MMU during address translation. The MC68040 is similar to the MC68030, except that the term *bus error* has been replaced with the term *access error,* and the signal `/BERR` has been replaced with the signal `/TEA` (this Note uses pre-MC68040 terminology).

What Is a Bus Error Handler?

A bus error handler is the exception handler for the bus error exception and is designated by offset $08 in the Exception Vector Table. The bus error handler is responsible for the recovery from the conditions that led to the bus error. Depending on the cause of the bus error, recovery may be either simple or extremely complex. In the case of a page fault, the recovery process entails loading the desired page into physical memory and updating the MMU Page Tables. In the case of a NuBus try-again-later acknowledgment, the recovery process is to simply retry the bus cycle that caused the bus error.

From the processor's perspective, there is, at any one time, only a single handler for all bus error exceptions. That handler is the one designated by offset $08 in the Exception Vector Table.

From the system programmer's perspective, there are many pieces of software that work in unison and form the bus error handler. Each of these individual pieces exists in an ad hoc linked list called the Bus Error Handler Chain and must follow certain installation, removal, and invocation rules to ensure proper system behavior.

How Does the Bus Error Handler Chain Work?

The bus error handler chain is rooted in location $00000008. This location contains a 32-bit pointer to the first handler. Each handler is responsible for maintaining the links in the chain. Invocation of the bus error handler chain works in different ways depending on the absence or presence of virtual memory (VM).

If VM is present, the processor's VBR (vector base register) points to a special Exception Vector Table which must **never** be modified. When a bus error occurs, the VM bus error handler is invoked and determines whether or not the bus error is a page fault. If the bus error is a page fault, VM takes the appropriate actions. If the bus error is not a page fault, VM invokes the first entry in the bus error handler chain.

In non-VM systems, the VBR points directly to location $00000000; therefore, the first entry in the bus error handler chain is invoked directly by the bus error exception processing performed by the processor.

In either case, the techniques for installing and removing an entry in the chain are identical. The only difference is that when VM is running it gets first crack at all bus errors.

What Is the Model for When Bus Errors Occur . . . And Who Handles Them?

The only bus errors that are expected during execution of application or Toolbox code are caused by virtual memory page faults (if VM is running). As a general rule applications and the Toolbox should **not** be directly accessing hardware that can cause bus errors. There may be cases when hardware diagnostic applications need to install a bus error handler, but these should be very rare and they should follow the same guidelines that drivers must follow. The reason for this is that MultiFinder does not switch the bus error vector, so during a minor switch there is no way to know if the correct vector is in place. Applications should not install handlers into the bus error handler chain because the Process Manager does not context switch entries in the chain during its application-level context switches.

Certain parts of the operating system expect bus errors under infrequent and well-controlled circumstances. Each of these managers installs its handler in the bus error handler chain before the instructions that may cause bus errors and removes the handler after these instructions. The managers that currently handle bus errors are as follows:

- The Memory Manager handles some of the bus errors that could arise if it is passed corrupt handles or pointers.
- The SCSI Manager expects and handles bus errors relating to the Blind Transfer handshake on machines that implement that mechanism.
- The Slot Manager expects and handles bus errors during its accesses to configuration ROMs, because accesses to empty or nonexistent NuBus slots generate bus errors.

The only nonsystem software that should attempt to handle bus errors are NuBus device drivers. Typically, the only bus errors that happen during driver execution are those related to the device. Because device drivers, DCEs, and heap space allocated by drivers are all supposed to be in the system heap (which cannot be paged), no page fault bus errors should occur. I/O buffers that are passed to drivers through normal Device Manger entry points cannot be paged BEFORE the Device Manager hands the call off to the device driver. Drivers that access other memory in their caller's address space at interrupt level must cooperate with VM to ensure that those pages cannot be paged prior to receiving any interrupt that may access them. Page faults are not allowed during device driver interrupt handlers.

Adding Code to and Removing Code From the System Bus Error Handler

The technique for a NuBus device driver to install code in the bus error handler chain is fairly simple. Location $00000008 in the logical address space points to the first entry in the bus error handler chain. The installer must save the current contents of location $00000008 and place the address of its handler into location $00000008. To remove its code from the bus error handler chain, a NuBus device driver should simply replace the old value that was saved from location $00000008 during bus error handler installation.

When to Install Code to the System Bus Error Handler

A NuBus device driver should install a bus error handler around certain instructions or groups of instructions that access the NuBus device and could generate bus errors. The handler should be installed only when executing code that is part of the device driver. It is acceptable to enclose fairly large loops with a single install and remove operation rather than have an install and remove operation within the loop. It is **not** acceptable to install a handler when the driver is **opened** and remove it when the driver is **closed**.

What Should a NuBus Device Driver's Bus Error Handler Do?

First a word of caution regarding bus error handlers in general. The Exception Stack Frames generated by different M68000 family members under a given condition are quite different. Furthermore, the recovery mechanisms implemented by the handler must be fully aware of the limitations of the processor's `RTE` policy. For example, the MC68000 is not capable of finishing an instruction that was terminated by a bus error; the MC68010, MC68020, and MC68030 finish the instruction by resuming its execution at the point of termination; and the MC68040 can only finish some instructions by restarting their execution. One must therefore be sure that their bus error handler can handle any error that may occur on the '040. Techniques for writing bus error handlers are not contained in this Note, which discusses only how to register your handler with the system and how to pass along bus errors to other handlers in the chain.

The bus error handler must first ensure that the bus error is one that the bus error handler expects. To do this it must inspect the Exception Stack Frame pushed onto the system stack by the processor, for example by examining the `PC` or the Data Cycle Fault Address in the Exception Stack Frame. Extra caution must be used when examining the `PC` value because the `PC` in the Exception Stack Frame is not always the `PC` of the instruction that caused the bus error. This is true for the '030 and on the '040 the PC in the Exception Stack Frame will almost never be the PC for the instruction that caused the bus error. Due to caching on the '040, the instruction that caused the error may have been sitting in the cache for a long time before it was executed. This makes it hard to properly tell where the error originated. (This type of exception is called a name imprecise exception.)

If the bus error is one that is expected by the bus error handler, it should cure the problem and unwind. This can be done in any number of ways that are appropriate for the given driver and device. On one end of the spectrum, the bus error handler may simply use an `RTE` instruction to cause the bus cycle to be rerun whereas in other cases it may completely remove the Exception Stack Frame from the stack and jump to some other point in the driver. Exiting a handler by doing an RTE is not a good idea on the '040. You cannot ask the processor to rerun the faulted bus cycle on the '040 as you can on the '030. Due to pipelining, you may end up jumpling to a point in the driver where you were are not supposed to be.

If the bus error is one that is not expected by the bus error handler, then the course of action depends on whether the bus error happened during execution of an interrupt handler or noninterrupt-level code.

The noninterrupt-level bus error handling scheme requires that each driver's bus error handler pass the bus error exception along to its predecessor if it does not handle the bus error. This is accomplished by restoring the machine to the exact state at the time the driver's handler was invoked and by jumping to the handler address that was in location $00000008 at the time the handler was installed. The last handler in the chain is the system's handler that generates a system error from an unhandled bus error.

Interrupt-level bus error handling is rather different. These handlers should not chain to their predecessor because noninterrupt-level bus error handlers may be context sensitive and possibly nonreentrant. If a bus error happens at interrupt level in a given NuBus device driver's interrupt handler and that driver cannot handle the bus error, then the driver should call `_SysErr` and cause the machine to crash. If a NuBus device driver's interrupt handler causes a bus error and has not installed a handler in the system chain, the results are unpredictable and that driver is in error.

Why Should I Have Declaration ROMs?

As explained earlier, certain parts of the operating system expect bus errors under infrequent and well-controlled circumstances. One manager that currently handles bus errors is the Slot Manager. The Slot Manager installs its handler in the bus error handler chain before the instructions, which may cause bus errors, and removes the handler after these instructions. It expects and handles bus errors during its accesses to declaration ROMs (also known as configuration ROMs), because accesses to empty or nonexistent NuBus slots generate bus errors. The declaration ROM is an area on a NuBus expansion card that contains firmware that identifies the card and its functions, and allows the card to communicate with the computer through Slot Manager routines. However, communication with the Slot Manager is possible only if you configure your card's declaration ROM firmware properly.

To individuals who are contemplating using a workaround instead of declaration ROMs in their NuBus design: Don't do this! First of all, your design will not conform to the Macintosh implementation of the NuBus specifications. Second, it is very difficult to create a workaround for handling bus errors since you will need to be able to handle any error that may be returned to you. Lastly, bus error handlers will change in future versions of the Macintosh Operating System. When this happens, a design without declaration ROMs will very likely become incompatible with all Macintosh systems running the new software.

Creating a Workaround for Dealing With Bus Error Handlers

It can be very difficult to create a workaround for dealing with the bus error handler for all machines, especially for systems that contain a 68040. One particular point about the '040 is that when you receive a bus error, any writebacks that you may receive will now be pending. Due to the '040 pipeline and caches, when you receive a bus error there may be other writes waiting to be completed that are unrelated to the faulted one. These writes are called writebacks since they typically are cached in the data cache and may correspond logically to various instructions prior to the faulted instruction. It is very important that the bus error handler be able to take care of this. For an example of how MacsBug handles bus errors for the '030, there is a code snippet available on the *Developer Essentials CD Series* disc. There is no code available to describe how MacsBug handles '040 bus errors. The '040 is a much more complex machine. MacsBug handles bus errors in a completely cursory fashion, so it never gets into the complexities on the '040. Its only job is to display the first error that occurred, based on the stack frame's PC. Subsequent bus errors in the frame will likely cause a crash (that is, a hung machine).

Because we are unable to provide sample code to demonstrate how MacsBug handles bus errors on the '040, we have decided to go through all the complexities involved in creating a bus error handler for an '040 machine. You will see why it is a sane idea to include declaration ROMs in a NuBus design.

Getting a bus error handler installed is no real problem, but writing one to handle all CPUs is difficult. The '040 is particularly difficult since the handler must be able to resolve up to three pending writes that may be in progress at the time the bus error is acknowledged. This imposes many constraints on writing a bus error handler. If you have an init or a cdev, you can install a bus error handler when you call the board to do whatever it does. (You should always remove the handler, and not leave it installed, since that is likely to conflict with VM and the rest of the system.) You cannot ask the processor to rerun the faulted bus cycle, as you can on the '030. Your handler must repair the fault, emulate the access, and perform all writebacks. Performing the writebacks can be quite complex depending on the type. Firstly, the access address may be either physical or logical. This is dependent on VM. If you are running VM you will need to translate the physical address to its logical addresses. Secondly, the access data may be either memory aligned or register aligned. And lastly, performing a writeback may itself cause another bus error which will require that your bus error handler be reentrant so that a fault from a writeback can be handled. After handling the writebacks you will need to alter alter the PC so that you can properly return and go on to the next instruction. The only problem here is how to find where the PC should be reset to. The PC in the exception frame is for the instruction that was in progress at the time of the fault, that is typically several instructions past the actual faulting instruction. Therefore there is no way to restart the PC in a reasonable spot, since you cannot tell from the stack frame where the instruction starts (in every case). Remember, you get bus errors in the middle of instructions that can be many words long, and in the '040 case, you can actually get bus errors for things for which you do not have a PC. Such a case is presumably rare, but if you do not handle it you will get a further crash. It is possible to put in some heuristic method, but that will fail occasionally.

As you can see, it would be very difficult to create a workaround for dealing with the bus error handler for the '040. The best thing to do is avoid the need for a handler by including declaration ROMs in your NuBus design.

Debugging Hints for Writing a Bus Error Handler for the '040

If you have read the above information and still insist on writing your own bus error handler, the following are some final debugging hints that will help you out.

If you want to have first crack at bus errors, be sure to turn VM off. A '020, '030, and '040 machine has a VBR that points to the exception vectors and it is always active. On processor reset, the VBR is initialized to $0000 0000. The Macintosh Operating System will leave the VBR at $0. It may be changed by VM, debuggers, or any system code that wants to set up its own exception vector table at some other address. If VM is present, the VBR points to a special Exception Vector Table. When a bus error occurs, the VM bus error handler is invoked. (This is described in greater detail earlier in this Note.) The 68000, on the other hand, has no VBR. All of its exception vectors are always located at address $0.

Second, most of the problems we have seen have been cache related. The fundamental difference between the '040 and the '030 is the '040 caches. An instruction cache exists that may read in your code and cause it to run much faster than you intended it to. Therefore, if you are going to be using time-dependent code, be sure to turn the instruction cache off (using the Memory cdev or the _HwPriv trap).

Lastly, when you're having a hard time finding what is going wrong, you may want to use a RAM stuffing routine that takes pertinent numbers and puts them away for you to view later. This would be the best way to see whether your bus error handler is working correctly (for example, to see whether the VBR is pointing to the correct spot in the exception vector table, and so on).

Further Reference:

- *MC68030 User's Manual*
- *MC68040 User's Manual*

NuBus™ is a trademark of Texas Instruments.

Macintosh
Technical Notes

Developer Technical Support

#293: Most Excellent CD Notes

Written by: James Beninghaus

February 1991

This Technical Note discusses issues concerning the use of the AppleCD SC drive, the Apple CD-ROM device driver, and the Foreign File Access software extension.

Multiple CD-ROM Drives

Your application can get access to the driver by calling the Device Manager routine `_OpenDriver`:

```
osErr = OpenDriver("\p.AppleCD", &ioRefNum);
```

`_OpenDriver` returns the driver reference number for the AppleCD SC drive with the lowest SCSI bus number, and this is okay if you are going to control only one AppleCD SC drive. If you want to control or access more than one drive, you must compute the driver reference number yourself. You can use the following formula to compute SCSI driver reference numbers:

```
(32 + SCSI ID) - 1
```

The following code demonstrates how to open any AppleCD SC drive connected to a Macintosh. `OpenCD` takes a logical CD drive number, not a SCSI ID, as the input parameter `CDDrive`. A logical CD drive number of one refers to the AppleCD SC drive with the lowest SCSI ID connected to the Macintosh.

```
typedef struct WhoIsThereRec {
    ParamBlockHeader
    short          ioRefNum;
    short          csCode;
    struct {
        Byte       fill;
        Byte       SCSIMask;
    } csParam;
} WhoIsThereRec;

OSErr OpenCD(Byte CDDrive, short *ioRefNum) {

    auto    OSErr           osErr;
    auto    short           ioRefNumTemp;
    auto    short           CDDriveCount;
    auto    short           SCSIID;
    auto    WhoIsThereRec   *pb;

    pb = (WhoIsThereRec *) NewPtrClear(sizeof (*pb));
    osErr = MemError();
    if (0 != pb && noErr == osErr) {
        osErr = OpenDriver("\p.AppleCD", &ioRefNumTemp);
        if (noErr == osErr) {
            (*pb).ioRefNum        = ioRefNumTemp;
            (*pb).csCode        = csWhoIsThere;
            osErr = PBStatus((ParmBlkPtr)pb, false);
            if (noErr == osErr) {
                CDDriveCount = 0;
                for (SCSIID = 0; SCSIID < 7; ++SCSIID) {
                    if (BitTst(&(*pb).csParam.SCSIMask, 7-SCSIID)) {
                        ++CDDriveCount;
                        if (CDDrive == CDDriveCount) {
                            *ioRefNum = -(32 + SCSIID) - 1;
                            DisposPtr((Ptr) pb);
                            return noErr;
                        }
                    }
                }
                osErr = paramErr;
            }
        }
        DisposPtr((Ptr) pb);
    }
    return osErr;
}
```

Device Manager Routines and Parameter Blocks

The Apple CD-ROM driver does not conform to the design criteria of the Device Manager, so do not use high-level Device Manager calls, because they do not work. Mistakenly, status calls are used to change control settings of the device, and control calls are used to get status information of the drive. The high-level `Control` and `Status` calls do not anticipate this implementation and simply do not work; instead, use the low-level `_PBControl` and `_PBStatus` calls for all access to the drive.

Zero parameter blocks before using them. The unused bytes of the parameter blocks must be set to zero before you can use the parameter block in `_PBControl` or `_PBStatus` calls to the driver. Failure to zero the blocks results in the Device Manager calls returning an unexpected `ioResult` of `paramErr` (-50).

Binary Coded Decimal

The AppleCD SC driver communicates track numbers and absolute-minutes-seconds-frame addresses in what is known as Binary Coded Decimal (BCD) format. In BCD, every four bits are used to represent one decimal digit. When working with the AppleCD SC, the BCD values are only up to two digits in length, "99" tops. Table 1 illustrates some possible values and their representation in 2's compliment and Binary Coded Decimal form.

BCD		**Value**	**2's Compliment**	
Hex	**Binary**		**Hex**	**Binary**
0x01	00000001	1	0x01	00000001
0x09	00001001	9	0x09	00001001
0x10	00010000	10	0x0A	00001010
0x80	10000000	80	0x50	01010000
0x99	10011001	99	0x63	01100011

Table 1–BCD and 2' Compliment Value Comparison

To convert from a 2's Compliment number to a BCD number, take the value of the digit in the ten's place, store it in the leftmost four bits of a byte, then add to it the value of the digit in the one's place.

```
    Byte Decimal2BCD(Byte n) {
        return ((n / 10) << 4) + (n % 10);
    }
```

Converting from BCD to decimal requires multiplying the value in the leftmost four bits by 10 and adding the value of rightmost four bits to the result.

```
    Byte BCD2Decimal(Byte n) {
        return ((n >> 4) * 10) + (n & 0x0f);
    }
```

Block Addresses

Physical blocks on a Compact Disc are defined as being 2K bytes in size. Since the Macintosh operating system likes to work in 512-byte blocks, it sets the logical block size to 512 bytes. If you assume 2K blocks when using block addresses, you get into trouble. If you are going to access the drive using logical block addressing, either change the block size back to 2K or be sure the formula you use in conversion from an absolute-minutes-seconds-frames address to a logical-block address takes this difference into account.

Foreign File Access And The 'sysz' Resource

Large capacity ISO and High Sierra format discs can overload the default memory limits of Apple's current external file system software, Foreign File Access. (This is not a problem for HFS-formatted CD-ROM discs, since the File Manager deals directly with native volumes, bypassing the Foreign File Access software.)

Since unused memory reserved by Foreign File Access at INIT time cannot be reclaimed, Apple limited the amount of memory that is available to Foreign File Access. The Foreign File Access file contains a 'sysz' resource that reserves 71, 680 bytes in the system heap. If the 'sysz' is too small discs do not mount, but if it is too big it wastes precious memory. Using the default 'sysz' value, Foreign File Access cannot handle a CD-ROM with an extremely large number of files and directories. In addition, with multiple AppleCD SC drives connected, Foreign File Access may run out of memory if multiple ISO or High Sierra CD-ROM discs are mounted.

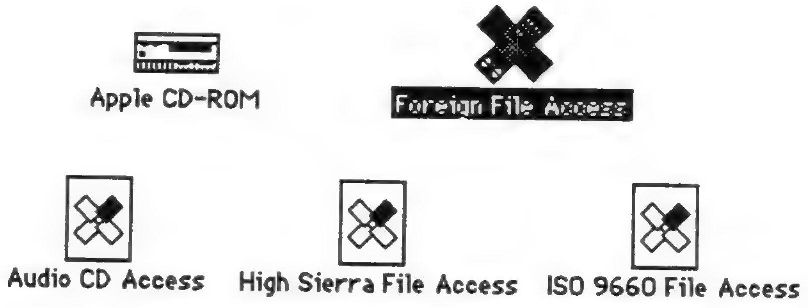


Figure 1–Apple CD-ROM Driver and Foreign File Access Software

Using ResEdit, you can experiment by changing the 'sysz' resource to find the optimal value for your disc's requirements. To avoid wasting valuable space in the System heap, increase this value incrementally until your disc mounts (after you reboot, of course). Asking your users to understand ResEdit or perform this operation is asking a bit much, so following is code upon which you could base a simple application to change the `'sysz'` value automatically for them. Remember that this application would need to be shipped separately (i.e., it is not accessible from the CD-ROM if the CD-ROM cannot be mounted).

This code assumes the creator and file type of the Foreign File Access file to be `ufox` and `INIT` respectively. It prompts the user to locate Foreign File Access using the Standard File Package routine `_SFGetFile`. This example does not allow the `'sysz'` value to be made smaller than Apple's default setting.

Note: You should not assume that Foreign File Access can be found in the System Folder; the Foreign File Access software resides in the Extensions folder when running System Software 7.0. Give the user the opportunity to find the file using a standard file dialog box.

```
char    *prompt    = "Find 'Foreign File Access'";
pascal Boolean FilterProc(HParmBlkPtr paramBlk) {
return 'ufox' == (*paramBlk).fileParam.ioFlFndrInfo.fdCreator ? false : true;
}

OSErr Modify_sysz(long size) {
    auto    OSErr      osErr;
    auto    SFReply    reply;
    auto    Point      where;
    auto    OSType     type;
    auto    short      resRefNum;
    auto    long       **sysz;
    osErr = noErr;
    SetPt(&where, 100, 100);
    type = 'INIT';
    SFGetFile(where, prompt, FilterProc, 1, &type, nil, &reply);
    if (reply.good) {
        resRefNum = OpenRFPerm(reply.fName, reply.vRefNum, fsRdWrPerm);
        osErr = ResError();
        if  (-1 != resRefNum) {
            sysz = (long **) Get1Resource('sysz', 0);
            osErr = ResError();
            if (nil != sysz) {
                if (0x00011800 < size) {
                    **sysz = size;
                    ChangedResource((Handle) sysz);
                    osErr = ResError();
                }
            }
            CloseResFile(resRefNum);
        }
    }
    return osErr;
}
```

Mixing Data and Audio

Any time the System, Finder, an application, or another code resource (e.g., XCMDs) accesses a disc, any sound being played from the disc is interrupted.

Further Reference:

- *AppleCD SC Developers Guide*, Revised Edition

#294: Me And My pIdle Proc (or how to let users know what's going on during print time...)

Written by: Pete "Luke" Alexander

April 1991

This Technical Note discusses how to defensively program a pIdle procedure to work with the majority of print drivers in existence today, and how to install it at print time.

Introduction

When using a pIdle procedure at print time, there are a few things that one should remember to be compatible with the printer drivers that are available today. This Technical Note discusses installing a pIdle procedure at the right time and the things to remember when writing one.

Installing The pIdle Proc

Let's start by installing the pIdle procedure at the right time. You must install your pIdle procedure into the print record before calling PrOpenDoc. If you do not install your pIdle procedure before your call to PrOpenDoc, the printer driver does not give the application's pIdle procedure any time. The following code fragments demonstrate installing the pIdle procedure in the right place:

MPW Pascal

```
<< more print loop would appear above, see Tech Note #161 for details >>

{** Install a pointer to your pIdle proc into your print record. **}

PrintingStatusDialog := GetNewDialog(257, NIL, POINTER(-1));
thePrRecHdl^^.prJob.pIdleProc := @checkMyPrintDialogButton;

thePrPort := PrOpenDoc(thePrRecHdl, NIL, NIL);

<< more print loop would follow below, see Tech Note #161 for details >>
```

MPW C

```
<< more print loop would appear above, see Tech Note #161 for details >>

/** Install a pointer to your pIdle proc into your print record.  **/

PrintingStatusDialog = GetNewDialog(257, nil, (WindowPtr) -1);
(**thePrRecHdl).prJob.pIdleProc = checkMyPrintDialogButton;

thePrPort = PrOpenDoc(thePrRecHdl, nil, nil);

<< more print loop would follow below, see Tech Note #161 for details >>
```

For a complete printing loop that handles errors at print time and makes all of the right calls to the Printing Manager, refer to Technical Note #161, A Print Loop That Cares...

Things To Remember About pIdle Procedures

It is extremely important to design and code your pIdle procedure as defensively as possible, thereby making sure that it works with as many printer drivers as possible. This section details a few things to remember when creating pIdle procedures.

Saving And Restoring The Current Port

It is extremely important to save the printer driver's `GrafPort,` upon entry to your pIdle procedure and restore it upon exit. Why? If you do not, the printer driver would draw into the `GrafPort` of your dialog box instead of it's `GrafPort`, which will cause some bad results. To save the printer's `GrafPort`, you should call `_GetPort` when entering your procedure. Before you exit your procedure, you would call `_SetPort` to set the port from your dialog box back to the printer driver's `GrafPort` (i.e., the one you saved with `_GetPort`).

Saving And Restoring The Printer Driver's Resources

If the application changes the resource chain within it's pIdle procedure, you want to save and restore the printer driver's resource chain. Why? Some printer drivers assume that their resource chain does not change, but this may not be true when the driver calls the pIdle procedure installed by the application at print time. To accomplish this task, call `_CurResFile`, saving the ID of the printer driver's resource file at the beginning of your pIdle procedure. When you exit from your pIdle procedure, restore the resource chain back to the printer driver's resource chain with a call to `_UseResFile`.

At this point, you might be wondering what might change the resource chain. If you called _OpenResFile or _UseResFile (anything that would change the value of the low memory global TopMapHdl) within the application's pIdle procedure, the chain would be changed. If you are not changing the resource chain, these calls would not be needed.

Handling Errors From Within A pIdle Procedure

You should avoid calling `PrError` within your pIdle procedure; errors that occur while it is executing are usually temporary, and serve only as internal flags for communication within the printer driver—they are not intended for the application. If you absolutely must call `PrError` within your idle procedure, and an error occurs, never abort printing within the idle procedure itself. Wait until the last called printing procedure returns, then check to see if the error still remains. Attempting to abort printing within an idle procedure is a guarantee of certain death.

Canceling Or Pausing The Printing Process

If you install a procedure for handling requests to cancel printing, with an option to pause the printing process, beware of timeout problems when printing to the LaserWriter. Communication between the Macintosh and the LaserWriter must be maintained to prevent a job or a wait timeout. If there is not any communication for a period of time (over two minutes), the printer times out and the print job terminates due to a wait timeout. Or, if the print job requires more than three minutes to print, the print job terminates due to a job timeout. Since, there is not a good method to determine to what type of printer an application is printing, it is probably a good idea to document the possibility of a LaserWriter timing out for a user who chooses to select "pause" for over two minutes.

Some Printer Drivers Do Not Support pIdle Procedures

Some printer drivers do not support pIdle procedures, as they prefer to handle the pIdle procedure in their own manner without giving an application's pIdle procedure any time. This situation should not be a problem as long as you do not assume that your pIdle procedure is always called at print time. Therefore, you should only create your pIdle procedure to display the dialog box and respond to a user pausing, continuing, or canceling a print job.

Conclusion

When installing your pIdle proc, it must be installed before the application calls PrOpenDoc. You want to make sure that you save and restore the GrafPort , upon entry and exit of your pIdle procedure, to make sure that the printer driver will image into the correct port during the print job. Finally, if you are changing the resource chain by calling _OpenResFile or _UseResFIle, you want to make sure that you save and restore the resource chain.

Further Reference:

- *Inside Macintosh*, Volume II, The Printing Manager
- Technical Note #161, A Print Loop That Cares...

#295: Feeder Fodder

Written by: Zz Zimmerman

April 1991

This Technical Note discusses the new Feeder button available in the 6.1, and 7.0 versions of the LaserWriter driver. This Feeder button mechanism allows developers to insert code into the LaserWriter driver to support a sheet feeder connected to a LaserWriter. This Note provides a description of the button, as well as information required to implement one.

Introduction

The LaserWriter driver now implements a standard method for handling sheet feeders. Most LaserWriter sheet feeders need a way to present a user with a dialog as well as a way to download the PostScript® code necessary to control the feeder. In the past, most manufacturers resorted to modifying the LaserWriter driver's code resources; however, this functionality is now possible without the need to patch existing resources in the driver—by adding three new resources.

When the LaserWriter driver notices these three special resources in its resource fork, it displays a Feeder button in the lower right corner of the Print dialog box. It is important to note that this feature is not provided for general application use, but rather only for developers of sheet feeders and other LaserWriter add-on devices. The button is always labeled Feeder, and there can be only one set of Feeder resources in the LaserWriter driver. Because of this restriction, you should not attempt to use this feature to implement anything other than a sheet feeder.

The first special resource contains code to implement the user interface of the feeder, and the other two contain the PostScript code required to drive the feeder. When an application calls the LaserWriter driver to display the Print dialog box, the driver looks for three resources of type `'feed'` and displays the Feeder button in the lower right corner of the dialog box if they are found. If no `'feed'` resources are available, it does not display the Feeder button.

When a user selects Print, the driver displays the standard Print dialog box with the Feeder button. If a user clicks on the Feeder button, the driver displays a dialog box in front of the Print dialog box, which allows the user to configure the feeder, then returns to the Print dialog box once the user confirms or cancels the feeder configuration. This feeder dialog box should **not** contain an option to print, as this could override choices made in the standard Print dialog box.

Implementation

To handle interaction with the user, you must install a resource of type `'feed' (ID = -8192)` into the LaserWriter driver with the code required to manage the dialog box. Like all Printing Manager code resources, this resource begins with a jump table, followed by the actual code. The code is implemented as a procedure that is passed a single parameter. This parameter is a rectangle defining the page size selected by the user. This page size is equivalent to the rPaper rectangle in the print record, meaning it defines the actual page size, not just the printable area. The rectangle is expressed in 72 dpi coordinates and has a negative origin.

Go Ahead and Jump

The jump table consists of a 68000 JMP instruction that jumps to the proper offset in the resource. In this case, there is only one routine, so the code starts immediately following the jump table. To make this step automatic, the jump table is created using a small assembly language header:

```
        IMPORT Feeder         ; Feeder is NOT defined here...
Start   MAIN   EXPORT          ; This is the main entry point for the linker.
        JMP    Feeder          ; The one jump table entry in this table.
        END
```

This example first imports the Feeder procedure, which can be defined externally in the language of your choice. Next is Start, the main entry point to the jump table. By passing this label to the link command, the jump table is located at the beginning of the resource. The next line is the actual jump table entry, and the END is required to end the assembly-language header. That's all there is to it. The only thing one should have to change in this code fragment would be the name of the routine to import.

The Real MacCode

Now that the jump table is complete, it needs some place to jump. Although MPW C and Pascal examples are provided in this Note, the code can be written in any language. As mentioned before, the code is implemented as a procedure that takes one parameter.

C Definition

In C, this looks like:

```
#include <Types.h>
#include <Quickdraw.h>
void FEEDER(Rect *r)
{
    <Code to present and handle dialog...>
}
```

Since the assembler converts all labels to uppercase, the name of the procedure FEEDER must be capitalized to match the case of the label in the jump table. If you are using MPW, you can use the assembler's CASE directive to prevent the assembler from capitalizing the labels. Since the rectangle is passed using the C calling convention (i.e., the caller strips the parameter), there is no need to declare the procedure as type Pascal. However, this convention does make things a little more interesting for the Pascal version:

Pascal Definition

If you are using MPW, you can use the Pascal compiler's `C` directive to define the Feeder procedure as using the C calling convention. This makes the definition look like this:

```
UNIT FeederSample;
    INTERFACE
        USES Types, Quickdraw;

    PROCEDURE Feeder(r: Rect); C;

    IMPLEMENTATION

    PROCEDURE Feeder(r: Rect);
    BEGIN
        <Code to present and handle dialog...>
    END;
END.
```

So this is straight forward. The procedure Feeder is defined as having one parameter (r), and the C directive is used so that the stack is handled correctly.

If you are using some other development environment that doesn't support the C directive, you have to do a little more work, making the definition look like this:

```
UNIT FeederSample;
    INTERFACE
        USES Types, Quickdraw;

    PROCEDURE Feeder;

    IMPLEMENTATION

    FUNCTION StealRectalParam: Rect;
        INLINE    $2EAE, 0008;        { MOVE.L        8(A6),(A7) }

    PROCEDURE Feeder;
    VAR
        r:    Rect;
    BEGIN
        r := StealRectalParam;

        <Code to present and handle dialog...>
    END;
END.
```

First of all, a unit is defined, and the proper interfaces are included. The definition of the `Feeder` procedure in the `INTERFACE` section is required to make the label available to external modules. In the `IMPLEMENTATION` section, one starts with the `StealRectalParam` function, which is used to get the rectangle passed by the Printing Manager without actually removing it from the stack. If you declared the rectangle as a parameter to the `Feeder` procedure, `Feeder` would remove the parameter before returning, then when the caller tried to remove the parameter again, the stack would be invalid and would cause a crash.

To solve this problem, define the `Feeder` procedure with no parameters. This way, the `Feeder` procedure leaves the parameter right where the caller left it. To get the parameter without removing it from the stack, use the `StealRectalParam` function, which moves the parameter from its normal location (off of `A6`) into the location pointed to by the stack pointer. Since `StealRectalParam` is a function, the stack pointer is already pointing to the return value. When StealRectalParam returns, the Feeder routine gets the rectangle parameter, without having removed it from the stack.

Tickled Link

Now you have the jump table and the code, but you still need to link them together. This step is pretty simple, but remember to specify the starting location of the jump table. It looks like the following:

```
Link -w -t feed -c ZzZz -rt feed=-8192 -m START -sg Feeder ∂
    Feeder.a.o ∂                # This file MUST be first.
    Feeder.p.o ∂
    "{Libraries}"Runtime.o ∂
    "{Libraries}"Interface.o ∂
    "{PLibraries}"SANELib.o ∂
    "{PLibraries}"PasLib.o ∂
    -o Feeder
```

First tell the linker to link the code into a 'feed' resource with an ID of -8192. Next, specify that the resource begins with the code at label START. This label was defined by the assembly-language used to generate the jump table. Finally, tell the linker to link all of the code into a single segment named Feeder. Obviously, the list of libraries and object files changes depending upon the language used, but the directives to the Link command should remain the same.

Well Fed

So that should be enough to get some code into the 'feed' resource. Now you need to actually control the feeder during the print job. To do this, you must use PostScript. Your driver should also provide a 'feed' resource of -8191 containing PostScript code. This code is downloaded by the LaserWriter driver prior to downloading the rest of the job. For those familiar with the 'PREC' 103 resource, the PostScript in the 'feed' resource is downloaded **before** the 'PREC' 103 code. Additional PostScript to be downloaded can be stored in 'feed' -8190. The PostScript code in the 'feed' resource should redefine (i.e., patch) the PostScript operators required to handle switching feeders. A likely candidate is the showpage operator called at the end of each page. As always, calling or redefining operators defined by the LaserPrep (md) dictionary is not supported. If your device is connected via the LaserWriter's serial port, you can license code from Adobe Systems, Inc. that makes it possible to access the serial port while the LaserWriter is connected over AppleTalk. For more information, contact Adobe at:

Adobe Systems, Inc.
1585 Charleston Road
Mountain View, CA 94043
(415) 961-4400

Once a user has confirmed the configuration from the dialog box, you can edit the PostScript code in the -8191 resource to reflect the choices made. However, when MultiFinder is active, you cannot add or change the size of resources in the LaserWriter driver. For this reason, you should pad the 'feed' -8191 resource to the maximum size. This padding can be done by adding spaces at the end. If you later need to resize the resource, you can simply overwrite some of the spaces. For more information on printer drivers under MultiFinder, see the *Learning to Drive* document, which is part of "Developer Essentials," and is available on AppleLink, the Apple FTP site, and the Developer CD Series.

You probably need to provide other resources along with the `'feed'` resources; for example, you need `'DITL'` and `'DLOG'` resources for the dialog box. This is okay, but you should be sure to pick unique resource types to avoid confusing the LaserWriter driver. In the case of a Feeder button, you are a guest in someone else's house. It would be wise to avoid rearranging the furniture.

When the LaserWriter driver actually opens the connection to the printer, it looks for `'feed'` resources -8191 and -8190. If they exist, they are downloaded. For those familiar with the `'PREC' 103` method of downloading PostScript code (refer to Technical Note #192, Surprised in LaserWriter 5.2 and Newer), the `'feed'` resources are downloaded **before** the `'PREC' 103` resource. In the case of background printing, the resources are copied into the spool file. Since `'feed'` resources -8191 and -8190 are automatically downloaded by the LaserWriter, they must contain PostScript code. The format of these PostScript resources is a string of ASCII characters without any length byte or terminator. The size of the string is determined by the size of the resource; there are no special size restrictions on these resources, and their only requirement is that they contain PostScript code. To make debugging easier, you should separate lines of PostScript using a carriage return character (13 or $0D hex).

Don't Feed The Print Monster

One last important note concerns the 6.1 version of the LaserWriter driver, shipped on the Macintosh Printing Tools disk included with the Personal LaserWriter LS and StyleWriter. In this version of the driver, the Feeder button will only work when Background printing is disabled. There is a problem with the driver finding the 'feed' resources when Background printing is enabled. This problem has been solved in the 7.0 version of the driver which should be used instead of the 6.1 driver as soon as it is available. Since there is no workaround for the problem, you don't really have to do anything except for possibly noting it in your documentation. Any note should recommend upgrading to the 7.0 version of the driver as soon as possible.

Driving Miss Lasey

Now that you have the two or three `'feed'` resources, the big question is installation. How should you ship these things? There are two methods. The first method involves licensing the LaserWriter driver from Apple Software Licensing (SW.License on AppleLink). This method is only required for "turn-key" systems, where all installation is done for the user and you must ship the LaserWriter driver as part of your product. The second method, which is by and large preferred as it requires no licensing, is to ship your resources in an installer application. This application simply opens the LaserWriter file and adds the necessary resources.

Conclusion

So this should be all the information you need to implement the feed resources for your device. If you intend to drive a sheet feeder through the LaserWriter's serial port, be sure to contact Adobe Systems, Inc. for the most current implementation and licensing information. Although the Feeder button could theoretically be used for other purposes, it will always be labeled "Feeder" by the LaserWriter driver. Because of this consistency, developers should not attempt to extend its functionality beyond support for sheet feeders.

Further Reference:

- *PostScript Language Reference Manual*, Adobe Systems Inc.

- Technical Note #192, Surprised in LaserWriter 5.2 and Newer

PostScript is a registered trademark of Adobe Systems Incorporated.

Macintosh
Technical Notes

Developer Technical Support

#296: The Lo Down on Dictionary Downloading

Written by: Zz Zimmerman April 1991

This technical note discusses a method for downloading PostScript dictionaries automatically using the LaserWriter driver. It will also provide the format and use of the PREC(103) resource. It will also describe some problems with the now obsolete PREC(201) resource. If you are using PostScript dictionaries, or either of these resources, you should definitely read this note.

Introduction

Although many picture comments have been added to support the features of PostScript that are missing from Quickdraw, many developers have still taken to sending PostScript directly from their applications. As the use and complexity of this PostScript code increases, more and more developers are finding it necessary to define and use their own PostScript dictionaries. PostScript dictionaries are basically collections of variables and procedures that can be predefined, and accessed later. They are used to prevent conflicts between the symbols used by applications and those used by system software (such as the LaserWriter driver's LaserPrep dictionary). Unfortunately, because of the LaserWriter drivers habit of using the PostScript 'save' and 'restore' operators, there are problems with keeping a PostScript dictionary defined. PostScript definitions made by code sent with the print job (ie. sent between the calls to PrOpenPage/PrClosePage) will be lost the first time the LaserWriter driver calls 'restore'. There are a couple of solutions to this problem, but one that hasn't been documented before involves the use of the PREC(103) resource. If the LaserWriter driver finds a resource of type PREC with an ID of 103, it will download the PostScript code to the LaserWriter **before** performing the initial 'save' operation. This means that any definitions made by the PostScript code stored in the PREC(103) resource will remain defined for the duration of the print job, independent of the LaserWriter driver's calls to save and restore.

Caveats

The PREC(103) method is a great way to get a dictionary downloaded at print time. Unfortunately, this does not solve the problem for using dictionaries in export files like PICT. If you insert PostScript code into Quickdraw pictures, the system is not smart enough to record the PREC(103) code into the picture. Instead, you must record the dictionary using the standard PostScript picture comments (defined in Technical Note #91, Optimizing for the LaserWriter–PicComments). You should also use the appropriate PostScript structuring comments as defined by the *Adobe Document Structuring Conventions*. If you use the Adobe comments correctly, an application that is importing your picture will have the option of parsing for the procedure set comments, and extracting the dictionary definition to be placed in a PREC(103) resource at print time.

The next caveat concerns the use of multiple PREC(103) resources. At PrOpenDoc time, the LaserWriter driver makes one GetResource call to load the resource of type PREC with an ID of 103. Because the call is a GetResource call (instead of Get1Resource), the PREC can be stored in any open resource file. To avoid any conflicts, the resource should be stored in the resource fork of your application, or in the document file that is referencing the PostScript dictionary. Because the GetResource call is only made once, only the first PREC resource found by GetResource will be used. Any other PREC(103) resources will be ignored. As long is this resource is only used by applications, there is no problem since there can only be one application active at any particular time. If the resource is used by other elements of the system (ie. desk accessories, drivers, INITs), you can easily run into the problem of your PREC resource being ignored. The best solution to this problem is to only use the resource from within an application.

Since the PREC(103) resource is considered part of the print job, the definitions it makes are lost when the job ends (ie. when the LaserWriter receives EndOfFile from the Macintosh). Because of this, the code you place in the PREC(103) resource should not attempt changing any persistent parameters in the printer. The means avoiding the PostScript 'exitserver' operator. You should also avoid calling other routines that reset the current state of the printer (ie. initclip, initgraphics, etc.). Use of these operators will have a serious effect on Quickdraw operations that may be present in the print job.

When the PREC(103) resource was originally introduced, it had a cousin called PREC(201). PREC(201) was similar to the PREC(103) resource in that it allowed PostScript to be downloaded to the printer before the actual print job. The difference between the two resources was that the PREC(201) resource downloaded the PostScript code at the same time that it downloaded the LaserPrep dictionary, outside of the PostScript 'server loop'. Because of this, any definitions made by the code in the PREC(201) resource would remain after the current job. Like the LaserPrep dictionary, the dictionary downloaded in PREC(201) would remain until the LaserWriter was rebooted (ie. powered off then on again). Although this feature was useful in some situations, it did have its problems. Not the least of which was the valuable printer memory consumed by the dictionary that was downloaded. Since the dictionary remained after the job that required it, subsequent jobs had less memory available to them. The only way to reclaim the memory was to reboot the printer, and this was not obvious to naive users. The other problem was introduced when background printing became available. With background printing enabled, the LaserWriter driver could no longer count on the PREC(201) resource always being available. Since you could no longer store the resource in the LaserWriter driver (because of the LaserWriter driver being MultiFinder compatible - see *Learning To Drive* for more information), it has to be stored in a separate resource file. This made it virtually impossible for the LaserWriter driver to find the resource when it was required. For this reason, the PREC(201) resource is only downloaded when background printing is turned off.

Needless to say, we don't recommend the use of features that only work in certain situations, so the PREC(201) resource is now considered unsupported and obsolete. If you are using the PREC(201) resource, you should be able to revise your application to use the PREC(103) resource instead, with only a small performance penalty. On the bright side, the PREC(201) resource will continue to be supported in the foreground through the 7.0 version of the LaserWriter driver, and most likely, until the new printing architecture becomes available, giving you plenty of time to revise...

Implementation

The PREC(103) resource can be implemented by simply creating the resource with ResEdit or Rez, and then storing it in an open resource file at print time. In the case of ResEdit, you should create a new resource of type PREC with an ID of 103. You should then open the new resource using the resource template for string ('STR ') resources. You can then type your PostScript code directly into the resource.

If you would rather keep your PREC definition as a Rez source file with the rest of your project, you can do this by simply defining the PREC resource type at the top of the file, followed by an instance of the PREC resource. Consider the following Rez source code:

```
/* First the resource type definition: */
type 'PREC' {
      string;
};

/* Now the real resource definition: */
resource 'PREC' (103) {
      "userdict /mydict 50 dict def";
};
```

We begin by defining the resource type as being a string. We then define an instance of the resource with an ID of 103. Finally, we define the contents of the resource. The PostScript code above basically defines a dictionary named mydict within the userdict dictionary. The mydict dictionary is defined as having a maximum of 50 elements. Consult the PostScript Language Reference Manual for more information concerning legal operations on dictionaries.

Conclusion

The PREC(103) is a simple, efficient way to download a PostScript dictionary at print time. It does not solve the problem of exporting PostScript that references a dictionary into file formats such as PICT, but it can help. Applications can be revised to extract PostScript dictionary definitions from files such as PICT and download them at print time using the PREC(103). It should be noted however that this is not automatic, the application must parse the picture to get this functionality. Finally, the PREC(201) resource can only be supported when background printing is disabled, so it is now considered obsolete, and use of it is unsupported.

Further Reference:

- *PostScript Language Reference Manual*, Adobe Systems Inc.
- *Adobe Document Structuring Conventions*, Adobe Systems Inc.
- *LaserWriter Reference Manual*, Addison-Wesley

PostScript is a registered trademark of Adobe Systems Incorporated.

Macintosh
Technical Notes

Developer Technical Support

#297: Pictures and the Printing Manager

Written by: Zz Zimmerman April 1991

This technical note described some problems and features of using Quickdraw pictures with the Printing Manager. In general, if your application prints Quickdraw pictures, you should read this note.

Introduction

Most applications support Quickdraw pictures to some degree. They will allow you to import or export picture files, as well as using the PICT resource format on the clipboard to support Cut & Paste with other applications. Unfortunately, there are some problems that occur with pictures at print time, and that's what I want to cover here.

You PICT When?

One of the problems that comes up at print time is the use of picture comments. Some applications store their data in a native format, and only create pictures at print time to enable the use of picture comments. For each page of the document, they open a new picture, record the Quickdraw calls that described the document, along with any picture comments they want to use, and finally close the picture. When this is done, they call DrawPicture to print the picture, and then start the whole process over again for the next page.

This method is supported and fully compatible with future system software, but is not required. The Printing Manager spools each page of a document into a Quickdraw picture. Since the Printing Manager already has a picture open, it is totally legal to send a picture comment (via the PicComment call) in between calls to PrOpenPage and PrClosePage without having them recorded in a picture. The Printing Manager has already replaced the StdComment procedure with its own anyway, so the PicComment call will be intercepted and supported correctly by the Printing Manager. If the only reason you are recording into pictures is so that you can use PicComments, you can avoid the overhead at print time by simply sending the comments directly.

Feeling PICT On?

Even if you aren't sending picture comments, you may still need to create a picture at print time. In general, you should try to create any pictures you need prior to calling PrOpen. This is because there is no way to predict how much memory a particular printer driver will require. Instead, you need to make as much memory available as possible. If you are creating pictures with the Printing Manager open, the chances are good that you are using memory you can't afford to waste.

If you need to create a picture with the Printing Manager open, and memory is not a problem, you should still be aware of some potential problems. First of all, keep in mind the Printing Manager receives data from the application by replacing the Quickdraw GrafProcs stored in the GrafPort returned by PrOpenDoc. One of these procedures is the StdComment procedure which is called each time the application calls the Quickdraw PicComment routine. Since the Printing Manager has these routines patched, creating a picture in the Printing Manager's GrafPort can cause problems. If you must create a picture between calls to PrOpenPage/PrClosePage, you should be sure to set the port to a standard Quickdraw GrafPort before calling OpenPicture. Any GrafPort that was created by Quickdraw (as opposed to the Printing Manager) will work fine.

If you do create a picture at print time, you may experience a syndrome we call 'floating picture comments'. That is, calls made by your application to the PicComment routine will be recorded in a different order than you made them. This will usually cause them to effect the wrong part of the picture, and lead to endless confusion. The best solution to this problem is to create any pictures that your application will need **before** opening the Printing Manager.

Scaling Pictures - Mountains from Mole Hills

Another problem is a basic problem with pictures that seems to show up more at print time. The problem concerns the scaling of pictures using the destination rectangle passed to DrawPicture. This method will work for most pictures, but problems arise with more complex pictures, and for pictures that contain text. The problem is the method that Quickdraw uses to scale the text stored within pictures. When scaling, Quickdraw tries to handle the text scaling intelligently by changing the size of the font being used, as opposed to just scaling the bits. Unfortunately, the widths used by bitmapped fonts are not always linear (ie. the 12 point width isn't exactly 1/2 of the 24 point width). Because of this, you can run into problems with lines of text getting slightly longer or shorter as the picture is scaled. In many cases, the error is insignificant, but if you are trying to draw a line of text that fits exactly into a box (a spreadsheet cell for example), you might be surprised to see the line of text extending beyond the box when the picture is scaled.

There can also be problems when using certain picture comments or imbedded PostScript. In the case of the TextCenter picture comment, you specify an offset to the center of rotation. This offset is usually based on the metrics of the font being used. If you scale the picture, Quickdraw decides to use a different font, and the offset you originally specified will be incorrect.

The easiest way to solve these problems is to scale the picture yourself. Especially if you are trying to scale by a large amount. For example, some applications create a picture at 72 dpi (ie. dots per inch), and then scale it to 288 dpi for printing by simply increasing the destination rectangle by 4x. This is asking a lot of the system, and will result in the text problems described above. Instead, you should either draw the picture into its original frame, and let the Printing Manager handle scaling it to the resolution of the device, or handle the scaling yourself by parsing the picture and playing it back opcode by opcode instead of calling DrawPicture.

One last thing to watch for when scaling pictures is integer overflows. It's usually pretty rare that you will overflow a coordinate when creating a Quickdraw picture, but it is not so hard to do when scaling a picture. For example, some applications will draw something offscreen to make sure the Printing Manager has configured the clip region and other related structures. They usually do this by moving the cursor to (-32767,-32767), and then draw a pixel. This works fine to initialize the Printing Manager, and the pixel isn't actually seen on the output. The problem occurs when you try to scale this picture. If you try to make it bigger, Quickdraw will adjust to coordinate (-32767,-32767) which will end up overflowing. The only way to solve these problems is to look for these kinds of operations in the picture before trying to scale it with DrawPicture.

Pictures Within Pictures–Is Nesting the Best Thing?

One cool feature of Quickdraw pictures is the ability to nest them easily. Basically, you can call OpenPicture, and then call DrawPicture with multiple pictures, and when you call ClosePicture, they will all have been recorded into one picture. Very cool. The problem comes when you start using the Begin/End form of picture comments. There are some comments like PostScriptBegin/PostScriptEnd, and TextBegin/TextEnd that have a begin comment that is followed by an end comment. When using these comments, it is very important to make sure that you have an end for each begin that you have sent. If the nesting gets off, you will, at the least, get incorrect output, though it is more likely that the Printing Manager will actually crash. If your application is generating picture comments, it is very simple to make sure that you have an end for each begin. But when nesting a picture that you have imported from another application, it is important to know how its comments will interact with yours.

In most cases, you can simply call DrawPicture to render the picture to the Printing Manager and you don't have to worry. But if you are creating a picture for export, you may have to nest multiple pictures from multiple creators into the same picture to be exported. If this is the case, it is important to make sure that all of your begin comments have matching end comments **before** attempting to insert another picture. If this is done, you can insert the imported picture without having to worry about the comments it contains. If all of your begin/ends are matched, you can assume that the imported picture will be just as valid.

On the other hand, if you have a begin comment, and want to insert a picture before inserting the appropriate end comment, you must parse the picture to be inserted to make sure it is not using the same comment pair. If it is, and you insert it, you will have problems.

So make sure that all your begins and ends are matched, and don't try to insert other pictures between begin/end pairs of comments. If you find that you have to insert a picture between a pair of begin/end comments, you must parse the picture to be inserted to make sure that it does not use the same comments.

Penalty for Quickdraw - Clipping

Here's a subtle fact about Quickdraw pictures. If you call OpenPicture, and then record some drawing operations, and you don't explicitly specify a clipping region, Quickdraw will specify one for you. In fact, Quickdraw will use the last clip region stored in the GrafPort that you are using when you call OpenPicture. This has been a surprise to many a developer when they record a picture, and a big piece of it ends up missing when they draw it. This isn't specific to print time, it can happen on the screen too, but it happens enough that it's worth mentioning.

D' Resolution

If you've read Technical Note #275, Features of 32 Bit Quickdraw 1.2, you probably noticed the new font and resolution information. Basically, fonts are now stored in pictures by name, not by ID. This means that fonts stored in pictures will be displayed correctly on any Macintosh without fonts remapping to other faces. The other new addition to the picture format is horizontal and vertical resolution information. Applications that create pictures using the new OpenCPicture call will be able to tell Quickdraw the native resolution of the picture data. So if you're a scanner that is scanning at 200 dpi, you will be able to store your data at 200 dpi (instead of scaling it to 72 dpi first). When an application subsequently opens the picture, it can determine the picture's resolution and take the necessary steps to display it correctly (ie. scaling down for display on the 72 dpi screen, or scaling up for display on 300 dpi devices like the LaserWriter).

Conclusion

Quickdraw pictures can be used successfully at print time, if you avoid the problems described above. Although there is a little overhead required by some of the workarounds, most are very simple to implement, and will help you avoid future compatibility problems.

Further Reference:

- *PostScript Language Reference Manual*, Adobe Systems Inc.
- *LaserWriter Reference Manual*, Addison-Wesley

PostScript is a registered trademark of Adobe Systems Incorporated.

Macintosh
Technical Notes

Developer Technical Support

#298: Color, Windows and 7.0

Revised by: Guillermo A. Ortiz May 1992
Written by: Guillermo A. Ortiz January 1991

System 7.0 introduces a new look for the Macintosh Desktop. In order to implement those changes 'wctb' and 'cctb' resources have changed in both form and use; it is now up to developers to take the lead and help the new standard work. The task can be divided in two main areas: in most cases all developers have to do is to stick to the system resources in order to provide a homogeneous feel to the user; developers in this group need only make sure the old 'wctb's are disposed of and that all dialogs and windows are based on CGrafPorts.

The other case is more restricted and involves developers that need to use their own colors; these applications have to define the resources using the new templates and do a careful selection of the colors in order to not break the color scheme implemented by the system.

Changes since January 1991: Removed note about 7.0 beta. Added mention to GetGray and added reference to where to find the sample 'WDEF'

Introduction

The good news is that the mechanics of coloring windows through the use of 'wctb' resources is amazingly well documented in Inside Macintosh volume V, the bad news is that System 7.0 uses a new and completely different scheme for colorizing windows. The new method uses 'wctb' resources that are different than what is described in IM V in both their contents and use, and it is no longer recommended that applications provide their own 'wctb's or that they change system 'wctb' resources at all.

This change is not arbitrary. System 7.0 establishes a new user interface that not only presents a new and better-looking appearance for windows, but also enhances the user perception of function. The new look helps the user find the place to click in order to produce a certain result.

As with most of the rest of the interface, Apple has already done the research and testing for you, so let the system do the work, and you can focus on your application's code. Of course an application **can** replace the 'wctb' provided by the system, but the results are bound to produce less-than-desirable results.

Effects on existing applications

Note: 'cctb' resources are now tightly coupled to 'wctb's (especially for scroll bars) and therefore the discussion about the effects of the new scheme on old 'wctb's also applies to 'cctb's; the extent of the effect depends on the type of application.

Applications that directly access 'wctb' resources to custom color windows and controls and can only deal with the old resources will not work; these resources are different and the elements that correspond to the old parts perform new functions or are ignored. Directly accessing 'wctb' resources may cause system crashes and/or produce really ugly results.

Solution: Revise application and utilities that manipulate 'wctb's to take into account the new data structure.

The system will ignore old-style 'wctb' resources; as a result applications that provide their own pre-System 7.0 window color table resources will not get the colors they used to see. The system will use the default look for the windows. If new style resources are provided then the entries will be used according to the new scheme; chances are the results are not going to be as good as those obtained with the system colors.

Solution: Take away the old resources and get used to the new system colors, your users will appreciate that your windows are similar to those across the system. In the few cases where it make sense update your resources to the new templates.

Applications that carry their own 'WDEF' and 'CDEF' resources will not get the new nice looking windows provided by the system and although these applications should not experience problems since they are doing all the work themselves the result will be a negative one from the good user interface perspective. These applications will have windows with the old look when all others look modern.

Solution: Developers should revise their applications to include 'WDEF' and 'CDEF' resources that are compatible with the new color interface. As of this writing, sample code for 'WDEF' can be found on AppleLink in the following location:

"Developer Support:Developer Services:System Software:
More US System Software:System 7 Golden Master:Sys7WDEF"

Certain colors are counted on to produce the correct shades in this new color interface, applications that completely destroy the color environment of the system will cause interface problems for the user. In the few cases when the system can find colors that produce a similar shading effect, the system will use those colors and display windows using the color interface (although not the same as all the other windows since the colors are different). When the system can not come up with a reasonable alternative for the colors it needs it reverts to displaying black and white windows.

Solution: Developers should revise their applications so that they don't take over the color environment and don't leave the colors all screwed up when switching out. Do use the Palette Manager, don't blast color tables, and don't hog all the available colors. The key to happiness is moderation.

The facts Ma'am, just the facts ...

The new data structure for 'wctb' resources resembles the old format but more 'part' fields are now present, the part codes for the new 'wctb's are:

Part code:	Part it corresponds to:
0 wContentColor	Content area of the window
1 wFrameColor	Frame
2 wTextColor	Window Title color & Default Text color for Dialog buttons
3 wHiliteColor	Reserved
4 wTitleBarColor	Reserved

5	wHiliteColorLight	Used to produce colors in Tittle Bar stripes and for grayed text
6	wHiliteColorDark	Used to produce colors in Tittle Bar stripes and for grayed text
7	wTitleBarLight	Used to produce colors in Tittle Bar Background
8	wTitleBarDark	Used to produce colors in Tittle Bar Background
9	wDialogLight	Used to produce the colors in a dialog's beveled frame
10	wDialogDark	Used to produce the colors in a dialog's beveled frame
11	wTingeLight	Used to produce tinges in parts of windows
12	wTingeDark	Used to produce tinges in parts of windows

The colors in the windows are generated algorithmically using the colors in the 7.0 'wctb'. Most of the colors are shades between the light and dark colors. For example, the background color of the title bar is a shade in between wTitleBarLight and wTitleBarDark. The resulting color is obtained by calling GetGray, check Inside Macintosh VI (Chapter 17: Color QuickDraw) for details on this call.

'cctb' resources are also different; here are the part codes and their corresponding parts for 'cctb':

Part code:		Part it corresponds to:
0	cFrameColor	Frames controls
1	cBodyColor	Background color in buttons.
2	cTextColor	Interior text in buttons and legend for radio buttons and check boxes
3	cThumbColor	Reserved
4	cFillPatColor	Reserved
5	cArrowsColorLight	Used to produce colors in Arrows and scroll bar background color
6	cArrowsColorDark	Used to produce colors in Arrows and scroll bar background color
7	cThumbLight	Used to produce colors in Thumb
8	cThumbDark	Used to produce colors in Thumb
9	cHiliteLight	(corresponding to wHiliteLight)
10	cHiliteDark	(corresponding to wHiliteDark)
11	cTitleBarLight	(corresponding to wTitleBarLight)
12	cTitleBarDark	(corresponding to wTitleBarDark)
13	cTingeLight	(corresponding to wTingeLight) Affects 5-6 and 7-8 above
14	cTingeDark	(corresponding to wTingeLight)

... but how does it work?

In System 7.0 windows and scroll bars are drawn in color on a color device 8 bit deep or more (4 bit deep or more in gray scale devices) independent of the type of grafport, the design gives windows and scroll bars a 'gray' look with subtle color tints around the corners; these tinges are intended to give the user hints about the functions of the different parts.

When a window is active it will be drawn with the frame in `wFrameColor`, the title in `wTextColor`, and the Drag Bar, the Scroll Bars and all the gadgets (Grow, Zoom and Go-away boxes,) in a gray color with the edges showing the tints; note that in the context of this Technical Note gray color can be different than RGB gray (R=G=B), for example if the light color is red and the dark color is blue then the 'gray' result will be purple. It is also important to note that the exact gray result may not be available in the color table of the target device in which case a close equivalent is used, in the cases when there is no equivalent available then the system resorts to black and white (old style) windows.

When the window is inactive the frame is drawn in a grayed `wFrameColor` to indicate its disabled state; the Drag Bar, the gadgets and the scroll bar of the window are whited out and the title will be grayed out (using gray color to display text, not the dithered gray produced with a 50% pattern) based on `wHiliteColorLight` and `wHiliteColorDark`. When the gadgets of a scroll bar (thumb and arrows) are enabled they are drawn in gray with tinting (coordinated with the color theme used by the window!); when disabled the thumb disappears altogether and the arrows show in gray only with no tinges.

In keeping the overall scheme of color interface the background pattern of scroll bars has to be a gray pattern based on `cArrowsColorLight` and `cArrowsColorDark`; when the scroll bar is disabled (when no scrolling is necessary to show all the items in a window) then the scroll bar will be displayed in a solid gray. Don't confuse this grayed out state with unselected windows that present the scroll bars as well as the drag bar and all gadgets completely whited out.

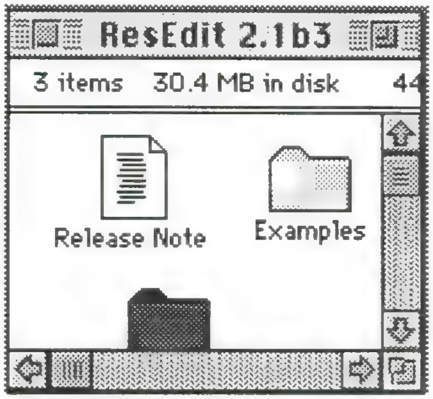


Figure 1 Active Window—Active scroll bars

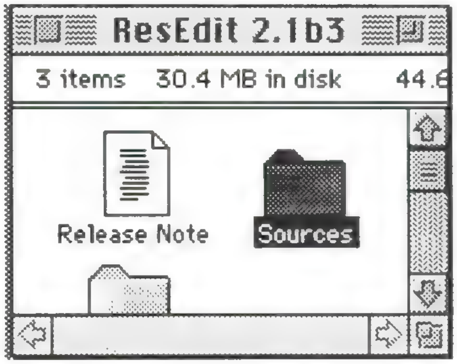


Figure 2 Active Window—Horizontal scroll bar disabled

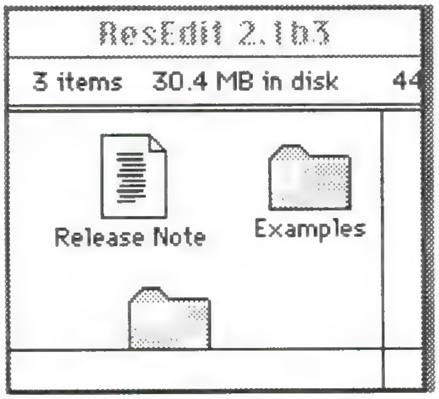
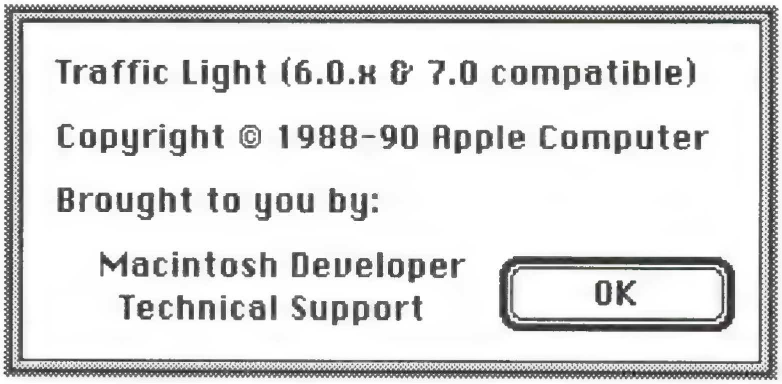


Figure 3 Inactive Window—Notice gray title

Dialogs and Alerts

Dialogs and alerts have also been colorized following the same theme as in windows, but instead of a tinged border Dialogs and alerts are displayed with a beveled border outlined with black; the bevel, with its spectrum of colors spreading between wDialogLight and wDialogDark produce a three dimensional effect. When a dialog becomes inactive the outline reverts to gray.



Traffic Light (6.0.x & 7.0 compatible)

Copyright © 1988-90 Apple Computer

Brought to you by:

Macintosh Developer
Technical Support

Regardless of the port (GrafPort or CGrafPort) dialogs and alerts are displayed using the shading scheme when the target device is set to 8 bits per pixel or more and colors, or when the target device is gray scale and set to 4 bits per pixel or more.

Buttons, Radio Buttons, Check Boxes and Text

Scroll Bars are not the only controls effected by 'cctb' resources, in general the names of the parts give a clear idea of what effect is produced by a given color, one area that is slightly different is Text; the text in buttons is drawn using `cTextColor` in a fashion similar to pre-System 7.0 systems, but when the button is disabled, the new system displays the text using gray color instead of using dithered gray like it did in earlier systems.

A gray color is used to draw the text of disabled buttons whenever the dialog is a CGrafPort and the depth of the target device is 2 bits per pixel or more. Dialogs based on old style ports will display disabled text using the old dithered gray.

The text associated with Radio Buttons and Check Boxes follows the same principles, text is the key to indicate the state (enabled or not) of Radio Buttons and Check Boxes since the body of Radio Buttons and Check Boxes is drawn using `cFrameColor` whether the control is enabled or not.

And you thought it would never end!

As always, all applications should refrain from non-friendly practices when dealing with the color environment; they should use the Palette Manager, and should never change color tables directly.

Further Reference:

- *Inside Macintosh*, Volumes V & VI, Color QuickDraw, Window Manager, Dialog Manager and Palette Manager.

Macintosh
Technical Notes

Developer Technical Support

#299: MultiFinder 6.0.x's Gross Anatomy

Written by: Paul Snively April 1991

This Technical Note discusses the programmatical interfaces to various of MultiFinder's heretofore undocumented capabilities under System 6.x.

Introduction

Historically MacDTS has refused to divulge considerable useful information about MultiFinder in the System 6.x world because the manner in which many capabilities were implemented was subject to change in a fashion that might have resulted in a change to the API. With the advent of System 7.0 looming large, it seems appropriate to document many of the gory details about MultiFinder at this time.

Process Management

The one distinct area that remained undocumented in *The Programmer's Guide to MultiFinder* and the various Tech Notes to date was process management. Everyone knew that it must exist at some level—after all, selecting "About the Finder" from the Apple menu showed the status of the applications currently running—but Apple wasn't forthcoming with any information about how such things might be done. Of course, System 7.0 provides process management, and the System 7.0 Process Manager is documented in *Inside Macintosh*, Volume VI. However, there are differences between System 6.x MultiFinder and the System 7.0 Process Manager that need to be explained.

The routines defined by the Process Manager in System 7.0 are:

```
GetCurrentProcess         GetNextProcess          GetProcessInformation
SameProcess               GetFrontProcess         SetFrontProcess
WakeUpProcess             LaunchApplication       LaunchDeskAccessory
```

This Technical Note will define interfaces to the equivalent functions in MultiFinder for the 6.x world, although launching in the 6.x world is not as well integrated as it is in System 7.0, so those differences will remain. Another difference is the way that processes can be identified; in 6.x only 16-bit PIDs are used, as opposed to the 64-bit Process Serial Numbers (PSNs) of System 7.0. Because of this, to facilitate the writing of code that works both in the 6.x and the 7.0 worlds, and to be consistent with the *Programmer's Guide to MultiFinder*, the 6.x interfaces will all begin with `MF` (e.g. `MFGetCurrentProcess`).

Data Structures

A central data structure in MultiFinder 6.x is the Process Info Record. In Pascal, it looks like this:

```
TYPE MFProcessInfoRec =
    RECORD
        ProcessState:              INTEGER;            { process state }
        ProcessID:                 INTEGER;            { the process id }
        ProcessType:               OSType;             { type of task (usually APPL) }
        ProcessSignature:          OSType;             { signature of task }
        ProcessVersion:            LONGINT;            { version of task }
        ProcessZone:               THz;                { pointer to minor zone }
        ProcessMode:               LONGINT;            { process' created mode }
        ProcessNeedSuspResEvts:    BOOLEAN;            { process expects suspend/resume events }
        ProcessBack:               BOOLEAN;            { can accept background time }
        ProcessActivateOnResume:   BOOLEAN;            { app activate/deactivate on suspend/resume }
        ProcessDad:                INTEGER;            { process id of my dad }
        ProcessSize:               LONGINT;            { size of his world }
        ProcessStackSize:          LONGINT;            { size of his stack }
        ProcessSlices:             LONGINT;            { # of times process has CPU }

        ProcessFreeMem:            LONGINT;            { amount of free memory in heap }
        ProcessName:               STRING[31];         { name of backing app }
        ProcessVRefNum:            INTEGER;            { vrefnum of app res file }
END;
```

In C, it looks like this:

```
struct MFProcessInfoRec {
    short          ProcessState;            /* process state */
    short          ProcessID;               /* the process id */
    OSType         ProcessType;             /* type of process (usually APPL) */
    OSType         ProcessSignature;        /* signature of process */
    long           ProcessVersion;          /* version of process */
    THz            ProcessZone;             /* pointer to minor zone */
    unsigned long  ProcessMode;             /* process' created mode */
    Boolean        ProcessNeedSuspResEvts;  /* process expects suspend and resume events */
    Boolean        ProcessBack;             /* can accept background time */
    Boolean        ProcessActivateOnResume; /* app will activate/deactivate on suspend/resume */
    short          ProcessDad;              /* process id of my dad */
    unsigned long  ProcessSize;             /* size of his world */
    unsigned long  ProcessStackSize;        /* size of his stack */
    unsigned long  ProcessSlices;           /* # of times process has CPU */

    unsigned long  ProcessFreeMem;          /* amount of free memory in heap */
    Str31          ProcessName;             /* name of backing app */
    short          ProcessVRefNum;          /* vrefnum of app res file as passed to MFLaunch */
};
```

An important address to know is `$B7C`, which is the address that contains MultiFinder's A5 value. (This value will be `$FFFFFFFF` if MultiFinder is not present.)

In Pascal, you can represent this with:

```
TYPE
    LONGINTPtr = ^LONGINT;
    INTEGERPtr = ^INTEGER;
```

```
CONST
    MFA5 = $B7C;
```

In C, you should use:

```
#define MFA5 0xB7C
```

There are two more values that will be important to us:

In Pascal:

```
CONST
    ProcInfoArray = $FFFFECFE;
    FrontPID = $FFFFF7CE;
```

In C:

```
#define ProcInfoArray 0xFFFFECFE
#define FrontPID 0xFFFFF7CE
```

These definitions are offsets from MultiFinder's A5, and are valid for Systems 6.0.5 and 6.0.7.

Note: While the A5 global offsets are probably valid in other 6.0.x versions of MultiFinder as well, they have not been tested in the others. Besides that, MacDTS recommends that for CPUs that feature sound input capability, System 6.0.7 be used, and that 6.0.5 be used for all other supported CPUs. Also, many developers use the so-called "Set Aside MultiFinder" that is provided with Apple's SADE debugger. The most recent such version of MultiFinder is 6.1b9. In MultiFinder 6.1b9, the value for `ProcInfoArray` is $FFFFF24A, and the value for `FrontPID` is $FFFFF23C.

Note: Determining the version of MultiFinder available is a pain, at least in Pascal. There is an OSDispatch selector to a routine that I'll call `MFGetVersion`, but it expects a pointer to a double-long (eight bytes), period. So in Pascal, you wind up with something like:

```
FUNCTION MFGetVersion(VAR String[8]: MFVersion): OSErr;

    FUNCTION InnerMFGetVersion(Ptr: Result): OSErr;
        INLINE $3F3C, $0002, $A88F;

BEGIN
    MFGetVersion := InnerMFGetVersion(@(Ord(MFVersion) + 1));
    MFVersion[0] := chr(8);         {Cheesy way to set the length byte}
END;
```

In C:

```
pascal OSErr MFGetVersion(char MFVersion[8])
    = {0x3F3C,0x0002,0xA88F};
```

With the Pascal version, the result will be a valid Pascal String[8]; with the C version, the result will be a char[8] that's been filled in. For MultiFinder 6.0.7 the result will be 'TWIT0110' and for MultiFinder 6.1b9, the result will be 'TWIT0120'.

Note: Having said all of that, MacDTS does not support the use of MultiFinder 6.1b9 for any purpose other than the use of the SADE debugger.

There are still three other useful numbers that will be needed:

In Pascal:

```
CONST
    LinkOffset = $18;
    PEntrySize = $6E;
    PIDOffset  = $02;
```

In C:

```
#define LinkOffset 0x18
#define PEntrySize 0x6E
#define PIDOffset  0x02
```

Armed with the `MFProcessInfoRecord`, MultiFinder's A5 pointer, the `ProcInfoArray` and `FrontPID` globals, and the `LinkOffset`, `PEntrySize`, and `PIDOffset` values, we can move on to the routines themselves.

MFGetCurrentProcess

`MFGetCurrentProcess` simply returns the PID of the currently executing process.

In Pascal the definition is:

```
FUNCTION MFGetCurrentProcess: INTEGER;
    INLINE $3F3C, $0013, $A88F;
```

In C, the definition is:

```
pascal short MFGetCurrentProcess()
    = {0x3F3C,0x0013,0xA88F};
```

`MFGetCurrentProcess` is useful in instances where the code calling it doesn't know what process is current, e.g. a desk accessory, driver, or trap patch might for some reason benefit from knowing which process is currently awake.

Note: The current process is not necessarily the same as the front process. The current process is the one whose A5 world is in place and whose code will be executed until the next context switch, either major or minor, occurs.

MFGetNextProcess

This routine is a natural adjunct to `MFGetFrontProcess`, as you can use it to cycle through all processes.

Unfortunately, this one isn't provided as a selector to OSDispatch; we have to roll our own. Fortunately, it's pretty simple with the magic constants given above.

In Pascal:

```
FUNCTION MFGetNextProcess(PID: INTEGER): INTEGER;

LocalPointer: Ptr;

BEGIN
    { Point to the ProcInfoArray }
    LocalPointer := @(Ord(LONGINTPtr(MFA5)^)+ProcInfoArray);
    { Point to the pointer to the next PEntry }
    LocalPointer := @(Ord(LocalPointer)+(PID*PEntrySize)+LinkOffset)
    { Dereference the pointer }
    LocalPointer := @LONGINTPtr(LocalPointer)^;
    { Get the PID from the new PEntry }
    MFGetNextProcess := INTEGERPtr(@(Ord(LocalPointer)+PIDOffset))^;
END;
```

In C:

```
pascal short MFGetNextProcess(short PID)
{
    Ptr    LocalPointer;
    LocalPointer = *MFA5+ProcInfoArray;
    LocalPointer = LocalPointer+(PID*PEntrySize)+LinkOffset;
    LocalPointer = *((Ptr *)LocalPointer);
    return(*((short *)(LocalPointer+PIDOffset)));
```

Note: MFGetNextProcess will always return a next process—don't expect there to be a point at which, when you call MFGetNextProcess, it will return zero. If you pass a PID to MFGetNextProcess, get the result, pass that to MFGetNextProcess, etc. you will find yourself in an infinite loop, as what seems like it should be the end of the line will simply point you back to the beginning.

MFGetProcessInformation

MFGetProcessInformation is the function that fills in a ProcessInfoRec, the definition of which is provided in the "Data Structures" portion of this Tech Note.

The definition in Pascal is:

```
FUNCTION MFGetProcessInformation(INTEGER: PID; VAR ProcInfoRec: ProcInfo): OSErr;
    INLINE $3F3C, $0017, $A88F;
```

In C:

```
pascal OSErr MFGetProcessInformation(short PID, ProcInfoRec *ProcInfo)
    = {0x3F3C,0x0017,0xA88F};
```

The majority of the ProcInfoRec is probably self-explanatory, with the likely exception of the ProcessState field. It can have one of the following values:

In Pascal:

```
CONST
```

```
    piStateReady = 1;
    piStateNull = 2;
    piStateBackRun = 3;
    piStateRun = 4;
    piStateUpdate = 5;
    piStateDebug = 6;
    piStateMoving = 7;
    piStatePuppet = 8;
    piStateSleeping = 9;
```

In C:

```
#define    piStateReady        1
#define    piStateNull         2
#define    piStateBackRun      3
#define    piStateRun          4
#define    piStateUpdate       5
#define    piStateDebug        6
#define    piStateMoving       7
#define    piStatePuppet       8
#define    piStateSleeping     9
```

These values should be considered informational and read-only; changing them in the ProcInfoRecord won't have any effect anyway.

MFSameProcess

The only reason that SameProcess exists in System 7.0 is that Process Serial Numbers (PSNs) are 64-bit quantities, and a language that shall remain nameless, but whose initial is "C," cannot directly compare structured data.

Since MultiFinder in the System 6.0.x world uses 16-bit Process IDs (PIDs), you can use the = operator in Pascal or the == operator in C to compare two PIDs. There is no such routine as MFSameProcess.

MFGetFrontProcess

Amazingly, MultiFinder 6.0.x doesn't have an OSDispatch selector for this! Luckily, it does maintain a global, so here goes:

In Pascal:

```
FUNCTION MFGetFrontProcess: INTEGER;
BEGIN
    MFGetFrontProcess := INTEGERPtr(@(LONGINTPtr(MFA5)^ + FrontPID))^;
END;
```

In C:

```
pascal short MFGetFrontProcess(void)
{
    return(*((short *)(*MFA5+FrontPID)));
}
```

This function is almost certainly most useful in conjunction with MFGetNextProcess and MFGetProcessInformation. You can call it, call MFGetProcessInformation, then MFGetNextProcess, and repeat to get information on all available processes, but heed the note under MFGetNextProcess—it doesn't eventually terminate, so in your loop you will have to compare the result of MFGetNextProcess with the result of MFGetFrontProcess, and terminate when they are equal.

MFSetFrontProcess

MFSetFrontProcess can be used to make a particular process the active process.

In Pascal, it looks like this:

```
FUNCTION MFSetFrontProcess(INTEGER: PID): OSErr;
    INLINE $3F3C, $0011, $A88F;
```

In C:

```
pascal OSErr MFSetFrontProcess(short PID)
    = {0x3F3C,0x0011,0xA88F};
```

Note: Use MFSetFrontProcess sparingly; as a generalization, the user should remain in control of which process is the active process. An example of an acceptable exception to this rule would be a backup utility that backs up a hard disk or a network at a particular time, such as 3:00 AM, and then wants to shut down the machine(s) gracefully. Since only the Finder can do that, it would be acceptable to use MFSetFrontProcess to make the Finder active, and then use PostEvent to fake a selection of "Shut Down" from the "Special" menu.

MFWakeUpProcess

Try as I might, I couldn't produce an analog to System 7.0's WakeUpProcess call in the System 6.0.x world. There is no OSDispatch selector for it, and neither is it something that can be easily reproduced by fiddling with some bits in the MultiFinder globals. Besides that, the effect that MFWakeUpProcess would have can be approximated with a little bit of forethought when you are designing your application. If your process is going to be communicating with another process somehow, for example, you should probably use a relatively small sleep parameter in your WaitNextEvent call so that your process can be somewhat responsive in terms of reacting to messages that it receives via IPC. You may wish to use a large sleep value initially, especially if your process has been switched into the background, and only decrease it upon receiving your first IPC message. When the IPC dialog has been terminated, you can then use a larger sleep value again. A little bit of software engineering can make up for this particular lack to a considerable degree.

MFLaunch

The important thing to note about MFLaunch is that it is distinct from Launch in that MFLaunch will not switch the thing being launched to the foreground. This is particularly useful, for example, to launch a background-only application whose sole *raison d'être*is to provide some service to the rest of the system.

Because Launch and MFLaunch are distinct, and don't really resemble System 7.0's Launch, no effort has been put into making MFLaunch resemble Launch in System 7.0.

In Pascal:

```
FUNCTION MFLaunch(String[31]: Name; LONGINT: Size; INTEGER: vRefNum; LONGINT:
TaskMode; LONGINT: StackSize): INTEGER;
    INLINE $3F3C, $0010, $A88F;
```

In C:

```
pascal short MFLaunch(StringPtr Name, unsigned long Size, short vRefNum, unsigned long
TaskMode, unsigned long StackSize)
    = {0x3F3C,0x0010,0xA88F};
```

MFLaunch will return the PID of the launched process, or an OSErr if there was a failure.

MFLaunch takes several parameters, but there are quite a few subtleties to be aware of with them and with MFLaunch in general.

First of all, if you've used Launch before, you probably found out the hard way that you need to set up the AppParmHandle appropriately first. The AppParmHandle is documented in Inside Macintosh V. I, pages 57-58.

Secondly, you must pass the Size parameter, and yes, that Size parameter does come from the SIZE resource. However, you can't just OpenResFile, get the SIZE resource, extract the value, and pass it verbatim. It has to be adjusted for both the required stack space and for the presence or lack of Color QuickDraw. So before passing the Size value, if you are on a Color QuickDraw system, add 32K. Also add in the default stack size for the system, which can be found in the low-RAM global DefltStack. But subtract the default stack size for a Macintosh Plus, which is 8K, because MultiFinder assumes that you need an 8K stack. Got that? Good.

The vRefNum parameter will probably be a WDRefNum that you'll have to create in order to specify the folder that contains the application. For that matter, since most applications expect the default folder to be the one they were launched from, you will probably need to create the working directory, and then do a SetVol on it before doing the launch.

The TaskMode parameter is simply the set of flags from the SIZE resource.

The StackSize parameter is whatever value is contained in the low-RAM global DefltStack without any of the tomfoolery necessary for the Size parameter.

Further Reference:

- *Inside Macintosh*, Volume VI, The Process Manager
- *Programmer's Guide to MultiFinder*
- Technical Note #158, MultiFinder Questions
- Technical Note #180, MultiFinder Miscellanea
- Technical Note #190, WDs & MultiFinder
- Technical Note #205, MultiFinder Revisited

Macintosh
Technical Notes

Developer Technical Support

#300: My Life as a PascalObject

Written by: Kent Sandvik & Mark Bjerke April 1991

This Technical Note discusses the `PascalObject` base class, used, for instance, with MacApp programming. The Technical Note describes how to write PascalObject derived classes that work with both Object Pascal and C++ code linking. It also describes the current restrictions and bugs with writing C++ code using `PascalObject` as the base class. This Technical Note is based on MacApp 2.0(.1), MPW 3.2 and MPW C++ 3.1.

Introduction

PascalObjects are useful. If you use `PascalObject` as the base class you are able to use the object libraries with both Object Pascal code and C++ code. MacApp 3.0 is written in C++ using `PascalObject` as the base class. Code written in Object Pascal is link compatible with C++ as long as any C++ objects which are accessed are derived from a PascalObject base class.

There are C++ semantics that will not work with Object Pascal, and there are semantic limitations with Object Pascal that the C++ programmer should be aware of if the code is to be used with Object Pascal.

An important issue is the class interfaces, C++ and Object Pascal. Any C++ language constructs that don't work in Object Pascal should not be present in the Object C++ header files. For instance there is no notion of `const` in Object Pascal, so a C++ declaration using `const` would be misleading because the value could be changed (from Object Pascal). Note that code inside the class methods [1] (that are not accessible for the class user) does not need to conform to Object Pascal limitations, with some exceptions which we try to cover in this Technical Note. Be careful when defining interface definitions for both Object Pascal and C++ use. Avoid any C++ syntax or semantics which can't be mapped to Object Pascal, if the intention is to produce libraries that will work with both C++ and Object Pascal.

PascalObjects behave like classes defined in Object Pascal, with the same kind of relocatable, handle-based objects and the same kind of method lookup tables. In the case of MacApp the TObject class is inherited from the PascalObject base class. Do not confuse PascalObjects with the HandleObject base class, even if both use handles they differ at the base class level.

[1] We use the Smalltalk terminology in this Technical Note, where *method* loosely corresponds to the C++ definition *member function*.

How to Write C++ PascalObject Classes

Constructors/Destructors in the PascalObject Domain

In the wonderful world of C++ programming the constructor takes arguments for values that are needed for the creation of the instance of the class. The C++ programmer is also able to pass arguments up to the parent classes if needed. In C++ the construction of a class starts with the base class, and each constructor down the inheritance chain is run.

This will not work with Object Pascal code. The construction of a PascalObject is usually an assignment of memory, and any possible initialization is done in a special method called IClassName (where ClassName is substituted to the class name, for example IMyApplication).

So if the C++ programmer assumes that everyone in the galaxy will use C++ notation for signalling information to the construction of the class, she/he is wrong. The policy is to create initialization methods for each class, and inside each method call the parent initialization method. This way all the initialization methods all the way to the top level are called. Note also that the order of calling base class constructors is implementation dependent, whereas in C++ the base constructors are called first, and the child constructors later. It usually makes sense to define this as well with PascalObject hierarchies, so the class library user could rely on the order of class initializations.

This is also true of destructors: instead of calling the destructor you need to define a method call Free (when using PascalObjects). You also need to call the method Free yourself, instead of relying as in the C++ world that the destructor will automatically work when the object goes out of scope.

Here's a simple MacApp example:

```
class TFooApplication: public TApplication
{
   public:
       virtual pascal IFooApplication(void);      // this is our "constructor"
       virtual pascal Free(void);                 // this is our "destructor"
// other methods and fields...
};

void TFooApplication::IFooApplication(void)
{
       this->IApplication(kFileType);      // call the base class constructor
       // do own stuff...
}

void TFooApplication::Free(void)
{
       inherited::Free();                  // call base class functions above
       // do own stuff...
}
```

pascal Keyword, virtual/OVERRIDE

Always define every method with Pascal calling conventions, as in the following example:

```
class TFoo: public TObject
{
   public:
       virtual pascal void Reset(void);
// ....
}
```

This means that you are able to call the function from Object Pascal. If you define the method virtual, and it's PascalObject based, then you are able to override the method from Object Pascal using the `OVERRIDE` keyword.

Function overloading as such does not work from Object Pascal, because Object Pascal does not have the notion of function name mangling.

Stack Objects (Objects on the Stack)

Object Pascal is not capable of defining objects on the stack . PascalObjects (as handles) are heap based. The compiler also complains if you try to define stack-based PascalObjects with C++.

A C++ example of a object declared on the stack is shown below:

```
void foo(void)
{
TDaffyDuck myDaffyDuck;                     // declared on the stack

       myDaffyDuck.ShootMeNow(kDuckSeason);
//     continue with the function…
}
```

private/protected/public in C++ and Object Pascal

C++ has access control of methods and fields inside classes, using `private`, `protected` and `public` as keywords. Field checks are done during compilation time, not linktime or runtime. Because the C++ modules are compiled separately from the Object Pascal modules, access control is not active from the Pascal code. It does not hurt to specify access control for C++ classes - quite the contrary - but Object Pascal code is able to access any method or function inside the C++ class.

So beware that any dependencies of the C++ class access control definitions will be broken under Object Pascal if you use PascalObject as the base class.

Str255 - Pascal Strings

Pascal strings are the common method for passing strings between multilanguage modules. (There are exceptions in the MacOS Toolbox.) If a method or function sends or receives a string, it should be declared as a Pascal String (`Str255`, `Str63`, `Str31…`). The MPW libraries have functions for changing, copying and comparing both Pascal strings and C strings (null terminated strings).

Breakpoint Information (%_BP and %_EP)

If you want to generate breakpoint information in the object files from C++, specify the '-trace on' flag to the C++ compiler, or use the new #pragma trace on and #pragma trace off switches inside the code. This only works with MPW 3.2 C and future releases. This is equivalent to the $D++ and $D- statements in Object Pascal.

Default Arguments

Object Pascal does not have the syntax for defining default argument values inside function prototypes. This means that you can't define default argument values in the FUNCTION and PROCEDURE methods in the Unit header files for Object Pascal (as you can do with C++ methods). If the user wants to change these default values there's no way to achieve this with Object Pascal.

Public Base Class

Classes that inherit from PascalObject should be defined with a public interface because the operator new is overloaded. An example of this looks like:

```
class TFoo: public PascalObject
{
// class contents
};
```

Inlining

Inlining of C++ methods works with the C++ header files, even if the semantics is not supported with Object Pascal header files. We assume that the inlining is used for C++ PascalObject class construction, where the classes are implemented in C++.

General C/Pascal Issues

Try to write code that works and functions well under both Object Pascal and C++. All the rules concerning Pascal and C code reusability are true for writing C++ and Object Pascal object libraries for as well. For instance avoid function and variable names with changes in capital case only, for example Foo and foo are identical function names under Pascal.

Also try to use the new "call by reference" notation of C++ (& - resembles VAR in Pascal) when passing references to variables to functions, instead of using the normal pointer notation. This way developers can write similar looking code for calling functions with values.

You also need to create Unit header files for Object Pascal use with the class definitions in Object Pascal. Remember to define any interface constants in Object Pascal that are defined as enums in the C++ class.

Bugs and Limitations with PascalObjects

General

Please consult the latest MPW and C++ Release Notes for the latest information about known bugs and limitations.

Pure Virtual Functions

Pure virtual functions are allowed in C++ PascalObject hierarchies, as long as these functions are defined. The compiler will complain if the programmer tries to create an instance of an abstract class (the class that contains the pure virtual function) with new. However the linker does not like that the pure virtual function is not defined, so when the programmer links object modules with classes containing pure virtual functions which are not defined the linker will complain.

The following code example shows the problem and the workaround:

```
class TAbstract  : public PascalObject {   // pure abstract class
      virtual void Method() =0;            // define pure virtual
};

class TDerived   : public TAbstract    {   // not abstract because
      virtual void Method() {};            // of this definition (non-pure)
};

void TAbstract::Method()                   // you need to define
{                                          // this function for the linker
// dummy, this is abstract anyway
}

TAbstract* noWay;
TDerived*  okClass;

main()
{
      noWay = new TAbstract;               // compiler will complain!

     okClass = new TDerived;               // OK, not an abstract class

     return 0;
}
```

Pointers to PascalObject Members

You may not have pointers to PascalObject member functions if you are using the MPW 3.1 Link tool. You must use the new MPW 3.2 linker and a new MPW 3.2 C++ compiler for using this feature. The old way of doing method dispatch was broken, but it is fixed in the new optimized dispatch code. (See `UObject.a` in MacApp 2.0.1.)

Multiple Inheritance

Because PascalObjects method lookup is based on Object Pascal method lookup tables (instead of normal C++ vtables), Multiple Inheritance does not work with PascalObjects.

Problems with including PascalObject Runtime Support

This bug has to do with not including the call to `_PGM` which brings in the segment containing the `%_SelProcs` and the method tables used in PascalObject method dispatch. This is flagged (for including this runtime support) when CFront sees a use of a member function belonging to a (type derived from) PascalObject. If it compiles `main()` before it sees a use of the member function then the call to `_PGM` will not be included. Note that even a call to operator `new` inside of `main()` does not do the trick for PascalObjects with constructors because the constructor calls operator `new`, not `main()`.

The way to get around this is to invoke operator `new` on a dummy object with no constructor (much like anti-dead stripping code). Remember that this is only necessary in cases where there is no code before `main()` referencing a PascalObject method. Below is code which reproduces this problem. Note that the call to operator '`new`' in `main` normally would be enough except that class `foo` has a constructor.

```
class foo : PascalObject {
      public:
            foo(void);
            virtual void meth1(void);
};

void main()
{
      foo* afoo = new foo;
}

foo::foo(void) { ; }
void foo::meth1(void) { ; }

void non_member_func(foo* theClass)
{
      theClass->meth1();
}
```

`pascal` Keyword

The `pascal` keyword is broken in the specific situation where one attempts to call a C function which returns a pointer to a Pascal-style function. The C compiler currently misinterprets the C function as a Pascal-style function and the function result is lost.

Problems with returning Structs/Objects

Methods may not return structs/objects or anything that requires the C compiler to push and address for the called routine to copy return values. This will break the method dispatch which is expecting a handle (`this->` pointer) to the object as the last thing pushed.

Alignment Problems with Arrays

There is an alignment bug involving the size the C compiler calculates for certain PascalObjects and the actual size CFront allocates for such objects using operator `new`. Basically if an object has a multidimensional array of a byte sized quantity (char, Boolean, etc) whose total size in bytes is odd, then pad bytes are added by CFront and the C compiler and everything is fine. Now if you have two such arrays declared (see example) back to back, then CFront makes the mistake of not adding the pad bytes.This results in the C compiler accessing memory that is off the end of the object in question (since the `new` was done with the size parameter too small). For example:

```
class foo  {           // CFront generates size as 20 - C compiler
                       // uses 22
  char bytes1[3][3];
  char bytes2[3][3];
  short x;             // Access of this field falls off end of object
};
```

set_new_handler()

To give more control over memory allocation, you could define an extern pointer from `set_new_handler(_new_handler)` to be called if operator `new` fails. This is not supported in the operator `new` used for `PascalObject` because the code for operator `new` fails to make the call to the user handler through the function pointer set with `set_new_handler()`.

Call of the Wrong Member Function

There is a bug that involves calling the wrong member function in the case of PascalObjects whose names differ only in case (for example `class Foo` and `class foo`).

This example shows the problem:

```
class foo : public PascalObject          // note the all lowercase name
{
      public:
      virtual pascal void foobar(void);
};

class Foo : public PascalObject          // n.b Foo
{
      public:
      virtual pascal void foobar(void);
};

pascal void foo::foobar(void) {}
pascal void Foo::foobar(void) {}


main()
{
 foo *afoo = new foo;
 Foo *aFoo = new Foo;

 afoo->foobar();
 aFoo->foobar();                // Calls the wrong 'foobar()'
}
```

Information

For more inside information about PascalObjects, check the MacApp files `UObject.a,` which describes how method lookup is handled, and `UObject.Global.p,` which shows how `NEW` is implemented under Object Pascal.

Conclusion

Using PascalObjects as the base class for your class libraries will get you many Object Pascal programmers as new friends. If you use TObject (from MacApp) as your base class library (subclass of PascalObject), you get a lot of meta-information and meta-methods for free. And PascalObject classes are handle based, so you get less memory allocation problems on small memory configuration Macintosh computers.

Further Reference:

- Technical Note #265, Pascal to C: PROCEDURE Parameters
- Technical Note #281, Multiple Inheritance and HandleObjects
- MPW C++ 3.1 Reference
- MPW C++ 3.1 Release Notes
- *C++ Programming with MacApp* , Wilson, Rosenstein, Shafer, Addison&Wesley
- *The Annotated C++ Reference Manual,* Ellis&Stroustrup, Addison&Wesley

Macintosh
Technical Notes

Developer Technical Support

#301: File Sharing and Shared Folders

Written by: Jim Luther

August 1991

This Note describes modifications to the existing File Manager routines, `PBGetCatInfo`, `PBHGetDirAccess`, `PBHSetDirAccess`, `PBHSetFLock` and `PBHRstFLock`, when used on volumes prepared by Macintosh System 7 File Sharing.

Introduction

There are several differences between System 7 File Sharing and AppleShare 2.0.1. This Note describes what those differences mean when calling `PBGetCatInfo`, `PBHGetDirAccess`, `PBHSetDirAccess`, `PBHSetFLock` and `PBHRstFLock` on local volumes that return `bHasPersonalAccessPrivileges` to `PBHGetVolParms`.

Share Points, Shared Areas, Locked Folders and PBGetCatInfo

The first notable difference between AppleShare 2.0.1 and File Sharing is that File Sharing allows both folders and volumes to be exported or shared over an AppleTalk network (only volumes could be shared with AppleShare 2.0.1). A folder or volume can be shared by selecting the "Share this item and its contents" check box in the Finder's Sharing dialog. A folder or volume shared in this way is called a "share point" and its Finder icon (if it's a folder) is shown in Figure 1. The share point and all folders under it in the directory structure have access privileges and those access privileges can be set by the local user.

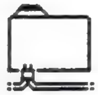


Figure 1–Folder that is a Share Point

The server's owner is a user with "All Privileges" and can remotely access all sharable volumes and folders on the Macintosh no matter what access privileges are set. The owner of an AppleShare 2.0.1 server is the server administrator. The owner of a File Sharing server is the owner of the Macintosh system as set by the Sharing Setup control panel. All other users of a server are considered regular users. Figure 2 shows the Finder icon of a folder that is a share point mounted by some regular user.

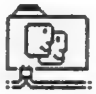


Figure 2–Folder that is a Share Point Mounted by a Regular User

Folders under a share point are already in a shared area and cannot be share points. However, those folders have access privileges so the visual feedback given by the Finder is the icon shown in Figure 3.

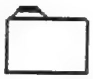


Figure 3–Folder in a Shared Area of the Folder Hierarchy

To allow applications to see share points and folders in shared areas, new bit definitions have been added to the `ioFlAttrib` bitmap returned by the File Manager call `PBGetCatInfo` when the information returned is for a folder. Bit 4 of `ioFlAttrib` is always set for folders. If a folder is a share point, bit 5 of `ioFlAttrib` is set. If a folder that's a share point is mounted, bit 3 of `ioFlAttrib` is set. If a folder is in a shared area of the folder hierarchy, bit 2 of `ioFlAttrib` is set. If a folder is locked, bit 0 of `ioFlAttrib` is set. Folders can locked or unlocked with the `PBHSetFLock` or `PBHRstFLock` calls. Figure 4 shows the `ioFlAttrib` bitmap for folders as returned by `PBGetCatInfo` under the System 7 File Manager.

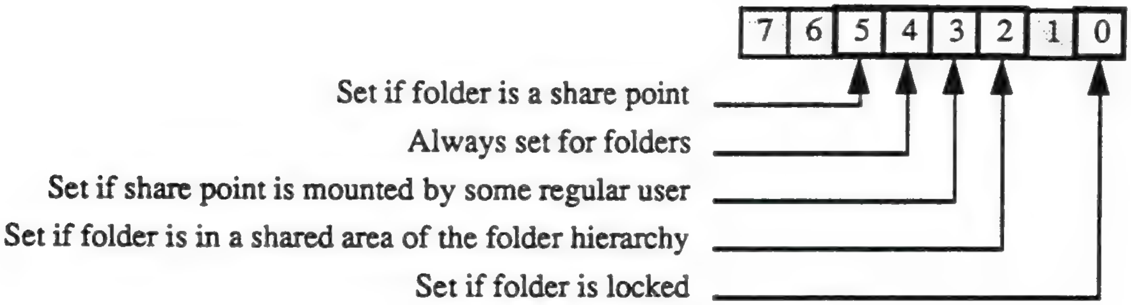

7
6
5
4
3
2
1
0
Set if folder is a share point
Always set for folders
Set if share point is mounted by some regular user
Set if folder is in a shared area of the folder hierarchy
Set if folder is locked

Figure 4–`ioFlAttrib` for a Folder

Note: These bits are READ-ONLY for folders. Do not try to set these bits with the `PBSetCatInfo` call.

Note: As noted in Inside Macintosh, Volume VI, `PBCatSearch` searches only on bits 0 and 4. The additional bits returned in `ioFlAttrib` by `PBGetCatInfo` cannot be used by `PBCatSearch`.

Shared Folders and Blank Access Privileges

Another difference between AppleShare 2.0.1 and File Sharing is that File Sharing supports a new user access privilege called blank access privileges. A folder with blank access privileges set ignores the other access privilege bits and uses the access privilege bits of its parent. On the local Macintosh, folders in a shared area default to blank access privileges (until set otherwise) and new folders created in a shared area are given blank access privileges. Folders created over AppleShare are given the same access privileges as the parent folder (or volume) and are owned by the user that created them.

Blank access privileges are useful because folders' access privileges now behave in a way which users expect them to. When a folder with blank access privileges is moved around within a folder hierarchy, it always reflects the access privileges of its containing folder. However, once the blank access privileges bit has been cleared for a folder, its access privileges "stick" to that folder, and remain unchanged no matter where the folder is moved.

Volumes that support blank access privileges have the `bHasBlankAccessPrivileges` bit set in `vMAttrib` longword of the volume parameter data returned by the `PBHGetVolParms` call. Folders with blank access privileges can be identified with the `PBHGetDirAccess` call. `PBHSetDirAccess` allows you to set blank access privileges. When bit 28 of `ioACAccess` is set, blank access privileges are set for a folder. The entire access privileges longword with the new bit for blank access privileges is shown in Figure 5.

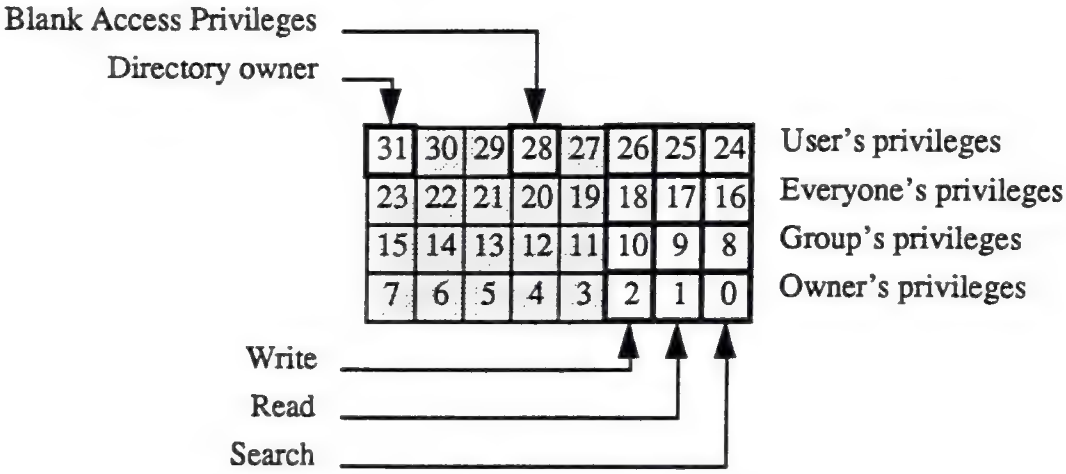


Figure 5–Access Privileges in `ioACAccess`

Note: Only the blank access privileges bit (bit 28) in the high byte of `ioACAccess` may be set when calling `PBHSetDirAccess`. You cannot set the directory owner bit or the user's privileges of a folder.

Note: The blank access privileges bit is not returned in the `ioACUser` field by the `PBGetCatInfo` routine.

Further Reference:

- *Inside Macintosh*, Volume IV, The File Manager
- *Inside Macintosh*, Volume V, File Manager Extensions In a Shared Environment
- *Inside Macintosh*, Volume VI, The File Manager
- *Inside AppleTalk*, AppleTalk Filing Protocol

Macintosh
Technical Notes

Developer Technical Support

#302: Help for Movable Modal Dialogs

Written by: James "im" Beninghaus

August 1991

This Technical Note describes the process by which an application can remap the Help Manager 'hmnu' resource while a movable modal dialog box is on the screen. The Help Manager handles the case for modal dialog boxes but punts in the case of movable modal dialog boxes. The following information will help you get the correct interface performance.

What's involved

The System 7 support for movable modal dialog boxes is limited to providing the new 'WDEF' variant. The rest of the implementation of movable modal dialog boxes is left to the application. Applications must provide handling for all events intended for a movable modal dialog box. This could be accomplished by calling the IsDialogEvent and DialogSelect Toolbox routines, or using other Toolbox routines such as FindWindow, BeginUpdate, DrawDialog, EndUpdate, TrackControl, TEClick.

How you process the events is up to you, but when it comes to appropriate balloon help the application must call the EnableItem, DisableItem and HMSetMenuResID Toolbox routines. The HMSetMenuResID is used before and after enabling or disabling the menus. HMSetMenuResID routine maps an alternative 'hmnu' resource to your menus.

The Systems 'hmnu' string resource

Listed here are two alternative 'hmnu' resources. The first one uses the same strings that the Help Manager shows when ModalDialog is called. The constant kHMHelpID is defined in the interface files BalloonsTypes.r, Ballons.h, and Balloons.p. In general it refers to the ID of various Help Mgr resources. In this case it selects a STR# resource in the System and the constants 31 and 32 refer to the string index within that resource. These strings are the ones the Help Manager uses when a Modal Dialog Box is on the screen.

```
resource 'hmnu' (256,"System Movable Modal Dialog hmnu") {
        HelpMgrVersion,
        hmDefaultOptions,
        0,
        0,

        HMStringResItem {               /* Missing items */
                0, 0,
                kHMHelpID, 31,
                0, 0,
                0, 0,
        },
        {
                HMStringResItem {       /* Menu Title */
                        0, 0,
                        kHMHelpID, 32,
                        0, 0,
                        0, 0,
                },
        }
};
```

An alternate 'hmnu' from your application

If you don’t want to display the same strings that the Help Manager displays for Modal Dialog Boxes, you can map in your own alternate 'hmnu' resource such as the following.

```
resource 'hmnu' (256,"Application Movable Modal Dialog hmnu") {
        HelpMgrVersion,
        hmDefaultOptions,
        0,
        0,

        HMStringItem {                  /* Missing items */
                "",
                "This item is not available because it cannot be used with"
                "the About box on your screen.",
                "",
                "",
        },
        {
                HMStringItem {          /* Menu Title */
                        "",
                        "This menu is not available because it cannot be used with"
                        "the About box on your screen.",
                        "",
                        "",
                },
        }
};
```

This resource is just an example. It’s up to you to define the contents of the strings including the internationalization issues.

Using the alternate 'hmnu' resource

After displaying the movable modal dialog box on the screen, the application should disable inappropriate menus and items and map in the alternate 'hmnu' resources.

```
        menu = GetMHandle(mApple);
        DisableItem(menu, 0);
        HMSetMenuResID(mApple, 256);
        menu = GetMHandle(mFile);
        DisableItem(menu, 0);
        HMSetMenuResID(mFile, 256);
        menu = GetMHandle(mEdit);
        DisableItem(menu, 0);
        HMSetMenuResID(mEdit, 256);
        DrawMenuBar();
```

Removing the alternate 'hmnu' resource

After removing the movable modal dialog box from the screen, the application must enable appropriate menus and items and unmap the alternate 'hmnu' resources.

```
menu = GetMHandle(mApple);
EnableItem(menu, 0);
HMSetMenuResID(mApple, -1);
menu = GetMHandle(mFile);
EnableItem(menu, 0);
HMSetMenuResID(mFile, -1);
menu = GetMHandle(mEdit);
EnableItem(menu, 0);
HMSetMenuResID(mEdit, -1);
DrawMenuBar();
```

Note: The previous fragments of code do not perform error checking. Well-behaved applications perform error checking whenever required. In these example the menu handle should be checked for a nil value before calling DisableItem and EnableItem.

Further Reference:

- *Inside Macintosh*, Volume VI, Help Manager

Macintosh
Technical Notes

Developer Technical Support

#303: Using a PurgeProc

Written by: Mensch

August 1991

This Technical Note discusses the use of the purgeProc field of an application's heap zone.

Introduction

Most applications will never need to use a purgeProc. However, if your application requires the ability to maintain purgeable handles containing data, or you need to have special notification when a certain handle is purged, a purgeProc might help you.

What exactly is a purgeProc?

The purgeProc, which is documented in very briefly in*Inside Macintosh*, Volume 2, page 2-23, is the mechanism which allows the Memory Manager to alert your application that it is getting ready to purge a given purgeable handle. This warning is given so that you can save the data, or note for any special reason that the data no longer exists. The purgeProc is passed the handle that is being purged. It is up to you to determine if any action should be taken on the handle. The Pascal interface to the purgeProc looks something like this:

```
PROCEDURE MyPurgeProc ( theHandle:Handle );
```

In C it would look like this:

```
pascal void myPurgeProc(Handle theHandle);
```

Each zone has its own purgeProc pointer in its zone header. Each time the Memory Manager prepares to purge a handle, it checks the zone header of the zone that the handle belongs to, to determine if your application has installed a purgeProc. If this field is not NIL, it calls the routine pointed to with the handle being purged. This occurs for each handle being purged (not every purgeable handle necessarily). When your routine is called, test the handle passed to be sure that it is a handle you care about, and then act on it. Keep in mind that all handles that pass through your purgeProc may not be expected, since your application can create purgeable handles in a few ways, like calling _HPurge on an existing handle, or loading a resource (the resource could have the purgeable attribute set), or by calling a routine that could load a resource.

Installing a purgeProc

You install your own purgeProc by setting the field in your zone header. Here is a sample routine that installs a purgeProc:

```
PROCEDURE InstallProc;

VAR     myZone:THz;

BEGIN
```

```
        myZone:=GetZone;                                { recover my applications zone }
        gOldProc:=myZone^.purgeProc;                    { save the old purgeProc }
        myZone^.purgeProc:=@myPurgeProc;
END;
```

You must be sure to follow these rules regarding what a purge proc must, can, and cannot do.

- Do not rely on A5 being set properly to your application's globals. (See Technical Note #208).
- Do not cause memory to be moved or purged.
- Do not open a file (but you can write to an already open file).
- Do not dispose of or change the purge status of the passed handle.
- Only use purgeProcs when absolutely needed.
- Avoid using purgeProcs if you are also using the SetResPurge(true) feature.
- Do write any data to a data file synchronously.
- Do preserve the contents of all registers except A0-A2/D0/D1.
- Do use the FindFolder feature of System 7 to locate the temporary folder on the user's hard disk if you are creating a temporary file to hold the contents of purged handles.
- Do test the handle state first to determine if the handle belongs to the Resource Manager, to weed out most handle purges you do not care about
- Do not take too long to determine if the passed handle is in need of purge notification (many programmers do not realize just how many purgeable handles come and go, or how often their purgeProc might be called for a single new handle).

Here is a pseudo code sample that illustrates one possible use of the purge warning procedure: saving the contents of the handle to a file before purging.

```
Procedure PurgeWarning(theHandle:myHType);
begin
  SetUpApplicationA5;      { see Inside Mac and Tech Notes for how to do this }
  IF BAND(hGetState(theHandle),$20)=0 then
    BEGIN {If we get here the handle does not belong to a resource}
      IF InSaveList(theHandle) then WriteData(theHandle);
    END;
  RestoreOldA5;
END;
```

Remember, the save file should probably be open at this point because opening a file can cause memory to move. You will have to maintain a save list to indicate purgeable handles that need saving.

NOTE: If you plan to use the SetResPurge(true) option that allows you to modify purgeable resources (not a normal thing for an application to do), don't patch the purgeProc pointer. If you do, remove your purgeProc, call SetResPurge, then re-install the purgeProc, being sure that it calls through to the Resource Manager's routine after it is finished.

Conclusion

PurgeProcs can be used when an application needs to better manage low memory situations, and easily take advantage of large memory conditions. Be very careful when using them, however, keeping in mind that you are in the middle of a Memory Manager routine when you are being called, and you may be called often.

Further Reference:

- *Inside Macintosh*, Volume II, Memory Manager chapter
- Technical Note #208, Setting and Restoring A5
- *Inside Macintosh*, Volume VI, Finder Interface chapter (FindFolder)

Macintosh
Technical Notes

Developer Technical Support

#304: Pending Update Perils

Revised by: C.K. Haun <TR> October 1991
Written by: C.K. Haun <TR> August 1991

This Technical Note discusses potential problems when pending update events for windows behind modal dialogs are not serviced. This note also documents some new System 7 Dialog Manager calls.

Changes since August 1991: Added note clarifying how to use the new calls, documented use of StdFilterProc in Interface.o,and corrected code errors.

Introduction

Modal dialog boxes have always caused some problems with windows behind dialog windows. Since the `ModalDialog` call makes an internal event call that bypasses your normal event loop you have always had the potential for not knowing that updates have occured for the other windows in your application when you are in a `ModalDialog` loop.

If you've ever written a filter procedure for a modal dialog, you've probably seen this for yourself. Your filter will get a continual stream of update events. These events are not for the dialog, but are for the window behind the dialog, which has not been updated since the modal dialog came up. Since the event has not come through your normal event loop you have probably not serviced the update since you are only concerned about events for your dialog, so it keeps getting resent. The only way for the update to stop is for the update region of the affected window to be cleared, by the Begin/EndUpdate calls in your drawing routine (see the discussion of update handling in Inside Mac I, the Window Manager chapter).

This situation is exacerbated by screen savers or Balloon Help in System 7. If a screen saver becomes active while a modal dialog is up, or if your user has Balloon help on and part of a behind window is obscured by a balloon, then an update event will be generated for the behind window, and you normally have no way to clear it.

The Update And Modal Dialog

Under System 7 (and in System 6 under MultiFinder), if there is an update event pending for your application, no other applications, drivers, control panels, or anything else will get time.

Updates pending for other applications do not cause the problem, they will be handled normally by the application in the background. But updates for the frontmost application *must* be serviced or the other applications will not get time.

This is a potential Bad Thing. Many pieces of code need time to keep living, to maintain network connections, or just to look good.

A simple example is the Clock desk accessory. Open the Clock DA, then launch an application that you know has a modal dialog. Position the clock so you can see it, and you'll notice that it refreshes its time count even while it's in the background.

Now make sure there is a document window open in the frontmost application. Turn on Balloon Help from the Help menu.

Open a modal dialog in the application (the About box in most applications will work). Now move the cursor over the window behind the modal dialog. A balloon will appear saying something like "This window is not active because a dialog box is up....", and a piece of the window will be blasted by the balloon. Now look at the Clock. It has stopped running. The window that got zapped by the balloon now has an update pending for it, that update is going through the ModalDialog trap, and not through the program's event loop, so it is not being serviced. Time stops for all other applications.

Note: This **only** happens if the update is for the same application as the dialog box. If you blast a window in another application (like the Finder) then that update will be processed normally.

Yuck, that's nasty!

You have two choices in your application to prevent this from happening. The first is to have no other open windows in your application when you open a modal dialog. Obviously, this isn't a realistic solution.

The second, saner, solution is to provide yourself a mechanism to refresh all your windows from within your modal dialog.

A filter procedure (described in the Dialog Manager chapter of Inside Mac volume I) is the proper tool to use to fix this problem. You'll need to add a simple filter procedure to every dialog or alert you bring up in your application. And, in most cases, it can be the same filter for every dialog, so it's not a great deal of extra code.

You're going to have to do a little preparation to do this. Your filter proc needs to have a way to call the drawing procedure for any of your windows. There are many ways to do this, dictated by the specific needs of your application and your own programming style. You may want to create a window control object that contains a pointer to your drawing routine, you may want to include the same check and dispatch you have in your main event loop, or use another method which you are comfortable with.

The simplest, bare bones method, would be to include a flag for your drawing procedure in your window record refCon, and have your drawing routine vector based on the value in the refCon, as shown here.

In MPW C

```
/* Window drawing proc, defined somewhere else */
Boolean MyDrawProc(WindowPtr windowToDraw)
{
Boolean returnVal = true;
/* switch off the value you've stored in your window earlier */
switch(GetWRefCon(windowToDraw)) {
       case kMyClipboard:    /* draw my clipboard */
       DrawMyClip(windowToDraw);
       break;
       case kMyDocument:     /* document content */
       DrawMyDoc(windowToDraw);
       break;
       default:              /* do nothing for anything else, to prevent drawing a window */
       returnVal = false;    /* that isn't mine */
       break; }
/* this return value is used to tell the Dialog Manager if you've handled the update */
/* or not when this is called from your filter.  In normal uses (i.e. in response to  */
/* an updateEvent in your main event loop) the boolean is unnecessary, but it doesn't */
/* do any harm */
return(returnVal);
}
/* install the flag when I create the window */
myWindowPtr = GetNewWindow(kMyWindowID,nil,(WindowPtr)-1);
SetWRefCon(myWindowPtr, (long)myDrawingProcFlag);
```

In your filter, the update handling would look something like this

```
if(theEventIn->what == updateEvt &&  theEventIn->message != myDialogPtr ) {
/* if the update is for the dialog box, ignore it since the regular ModalDialog function */
/* will redraw it as necessary */
return(MyDrawProc((WindowPtr)theEventIn->message));
/* go to my drawing routine, window will be redrawn if I own it */
}
```

In MPW Pascal

```
{ The function's result is used to tell the Dialog Manager if you've handled the update }
{ or not when this is called from your filter.  In normal uses (i.e. in response to  }
{ an updateEvent in your main event loop) the boolean is unnecessary, but it doesn't }
{ do any harm. The window drawing procedure is defined somewhere else. }

FUNCTION MyDrawProc(windowToDraw WindowPtr): BOOLEAN;

BEGIN
  CASE GetWRefCon(windowPtr) OF

    kMyClipboard:
      BEGIN
        DrawMyClipboard(windowToDraw);
        MyDrawProc := TRUE;
      END;

    kMyDocument:
      BEGIN
        DrawMyDocument(windowToDraw);
        MyDrawProc := TRUE;
      END;

    OTHERWISE
      MyDrawProc := FALSE;
   END; {CASE}
END;
```

Install the flag when you create a window:

```
  myWindowPtr := GetNewWindow(kMyWindowID, NIL, WindowPtr(-1));
  SetWRefCon(myWindowPtr, myDrawingProcFlag);
```

In your filter, the update handling would look something like this:

```
FUNCTION MyFilter(currentDialog: DialogPtr; VAR theEventIn: EventRecord; VAR theItem:
INTEGER): BOOLEAN;

{ if the update is for the dialog box, ignore it since the regular ModalDialog
{ function will redraw it as necessary }

BEGIN

  IF (theEventIn.what = updateEvt AND theEventIn.message <> currentDialog)
    BEGIN
      MyFilter := MyDrawProc(currentDialog);
    END;
  ...
  ...
 ...
```

If you do some, you have to do a little more....

The only down side to adding your own filter procedure to a dialog is that the Dialog Manager then assumes that you are handing more than just updates. Specifically, the Dialog Manager assumes that you are handling the standard "return key aliases to item 1" filtering. So, you need to write keystroke handling in the filter yourself.

The Dialog Manager in System 7 has some new calls you can make to ease the load on your program. These calls were created and tested too late in System 7's development cycle to be documented in Inside Macintosh, so they are presented here. They allow you to call on the services of the System to track standard keystrokes in your dialog.

> **NOTE:** You must call the standard filter proc (see GetStdFilterProc below) for these new calls to work properly. Automatic cursor tracking, default button bordering, and keystroke aliasing for OK and Cancel will only be active if you call the standard filter procedure. Also, these calls are System 7 specific. You cannot use them in previous system versions.

To make things even easier, MPW 3.2 and later contain glue code to allow you to call the standard filter procedure without calling GetStdFilterProc and dispatching to the procedure pointer returned. The glue routine is called StdFilterProc and is contained in the Interface.o file in the standard MPW libraries. The description of the call is included below. If you are not using the MPW development environment and do not have access to the MPW libraries from your development environment, you will of course have to get the procedure pointer and call it yourself.

```
New System 7 Dialog Manager call interfaces

MPW C
/* Returns a pointer to the Dialog Manager's standard dialog filter */
pascal OSErr GetStdFilterProc(ModalFilterProcPtr *theProc )
       = { 0x303C, 0x0203, 0xAA68 };

/* Indicates to the Dialog Manager which item is default.  Will then alias the return */
/* & enter keys to this item, and also bold border it for you (yaaaaa!) */
pascal OSErr SetDialogDefaultItem(DialogPtr theDialog, short newItem)
       = { 0x303C, 0x0304, 0xAA68 };

/* Indicates which item should be aliased to escape or Command - . */
```

```
pascal OSErr SetDialogCancelItem(DialogPtr theDialog, short newItem)
        = { 0x303C, 0x0305, 0xAA68 };

/* Tells the Dialog Manager that there is an edit line in this dialog, and */
/* it should track and change to an I-Beam cursor when over the edit line */
pascal OSErr SetDialogTracksCursor(DialogPtr theDialog, Boolean tracks)
        = { 0x303C, 0x0306, 0xAA68 };


/* This routine is included in the MPW 3.2 Interface.o library, and eliminates the */
/* need for you to have to dispatch to the ModalFilterProcPtr returned by GetStdFilterProc */
/* StdFilterProc will call GetStdFilterProc and dispatch to it for you */
pascal Boolean StdFilterProc(DialogPtr theDialog,EventRecord *theEvent,short *itemHit);
```

MPW Pascal

```
{ Returns a pointer to the Dialog Manager's standard dialog filter }
FUNCTION GetStdFilterProc(VAR theProc: ProcPtr ): OSErr;
        INLINE $303C, $0203, $AA68;

{ Indicates to the Dialog Manager which item is default.  Will then alias the return & }
{ enter key }
{ to this item, and also bold border it for you (yaaaaa!) }
FUNCTION SetDialogDefaultItem(theDialog: DialogPtr; newItem: INTEGER): OSErr;
        INLINE $303C, $0304, $AA68;

{ Indicates which item should be aliased to escape or Command - . }
FUNCTION SetDialogCancelItem(theDialog: DialogPtr; newItem: INTEGER): OSErr;
        INLINE $303C, $0305, $AA68;

{ Tells the Dialog Manager that there is an edit line in this dialog, and }
{ it should track and change to an I-Beam cursor when over the edit line }

FUNCTION SetDialogTracksCursor(theDialog: DialogPtr; tracks: Boolean):OSErr;
        INLINE $303C, $0306, $AA68 ;

{ This routine is included in the MPW 3.2 Interface.o library, and eliminates the }
{ need for you to have to dispatch to the ModalFilterProcPtr returned by GetStdFilterProc }
{ StdFilterProc will call GetStdFilterProc and dispatch to it for you }
FUNCTION StdFilterProc(theDialog: DialogPtr; VAR event: EventRecord; VAR itemHit: INTEGER):
BOOLEAN;
```

MPW Assembly

```
selectGetStdFilterProc                  EQU                 3
paramWordsGetStdFilterProc              EQU                 2

selectSetDialogDefaultItem              EQU                 4
paramWordsSetDialogDefaultItem          EQU                 3

selectSetDialogCancelItem               EQU                 5
paramWordsSetDialogCancelItem           EQU                 3

selectSetDialogTracksCursor             EQU                 6
paramWordsSetDialogTracksCursor         EQU                 3


_DialogDispatch       OPWORD        $AA68
              MACRO
              DoDialogMgrDispatch &routineName
              DoDispatch _DialogDispatch,select&routineName,paramWords&routineName
              ENDM

; Returns a pointer to the Dialog Manager's standard dialog filter
; FUNCTION GetStdFilterProc(VAR theProc: ProcPtr): OSErr;
;
```

```
              MACRO
              _GetStdFilterProc
              DoDialogMgrDispatch GetStdFilterProc
              ENDM

; Indicates to the Dialog Manager which item is default.  Will then alias the return key
; & enter key to this item, and also bold border it for you (yaaaaa!)
; FUNCTION SetDialogDefaultItem(theDialog: DialogPtr; newItem: INTEGER): OSErr;
;
              MACRO
              _SetDialogDefaultItem
              DoDialogMgrDispatch SetDialogDefaultItem
              ENDM

; Indicates which item should be aliased to escape or Command - .
; FUNCTION SetDialogCancelItem(theDialog: DialogPtr; newItem: INTEGER): OSErr;
;
              MACRO
              _SetDialogCancelItem
              DoDialogMgrDispatch SetDialogCancelItem
              ENDM

; Tells the Dialog Manager that there is an edit line in this dialog, and
; it should track and change to an I-Beam cursor when over the edit line
; FUNCTION SetDialogTracksCursor(theDialog: DialogPtr; tracks: Boolean): OSErr;

              MACRO
              _SetDialogTracksCursor
              DoDialogMgrDispatch SetDialogTracksCursor
              ENDM
```

Using these calls requires a little preparation on your part. After you create your dialog, you need to tell the Dialog Manager which items you want as the default and cancel items. The button selected as the cancel item will be toggled by the Escape key or by a Command-. keypress. The button specified as the default will be toggled by the return or enter key, and also will have the standard heavy black border drawn around it! The buttons will also be hilited when the correct key is hit.

The `SetDialogTracksCursor` call tells the Dialog Manager that you have edit lines in your dialog. When you pass a 'true' value to the `SetDialogTracksCursor` call the Dialog Manager will constantly check cursor position in your dialog, and change the cursor to an I-Beam when the cursor is over an edit line.
So the complete System 7 filter, incorporating update handling and new Dialog Manager calls, will look something like this;

MPW C

```
/* Before we go into a ModalDialog loop, do a little preparation */
myDialogPtr = GetNewDialog(kMyDialogID, nil, (WindowPtr)-1);
myErr = SetDialogDefaultItem(myDialogPtr,ok);    /* Tell the Dialog Manager that the OK */
                                                 /* button is the default */
myErr = SetDialogCancelItem(myDialogPtr,cancel); /* Tell the Dialog Manager the cancel */
                                                 /* button is the cancel item */
myErr = SetDialogTracksCursor(myDialogPtr,true); /* We have an edit item in our dialog,so  */
```

```
                                                  /* tell the Dialog Manager to change  */
                                                  /* the cursor to an I-Beam when it's */
                                                  /* over the edit line */
do {
    ModalDialog((ModalFilterProcPtr)myFilter, &hitItem);
    }while(hitItem != ok && hitItem !=cancel);

/* and your filter will look something like this */
pascal  Boolean  myFilter(DialogPtr  currentDialog,  EventRecord  *theEventIn,  short
*theDialogItem)
{OSErr myErr;
ModalFilterProcPtr standardProc;
Boolean returnVal = false;
WindowPtr temp;
if(theEventIn->what == updateEvt &&  theEventIn->message != currentDialog) {
                                          /* if the update is for the dialog box, ignore it */
                                       /* since the regular ModalDialog function
                                          /*will  redraw it as necessary */
        returnVal = MyDrawProc(theEventIn->message); /* go to my drawing routine */
  } else {
                                       /* it wasn't an update, pass it on to the */
                                       /* system filter */
        GetPort(&temp);                /* save the current port */
        SetPort(currentDialog);        /* and set to the dialog, this is necessary */
                                       /* to track the edit line */
                                      /* cursor change correctly */
/* NOTE: If you are using  MPW 3.2, there is a glue routine in the Interface.o library */
/* that will take care of the  details of getting and dispatching to the standard filter */
/* for you.  If you are not using MPW 3.2, you will have to call the standard */
/*  filter procedure yourself.  Both ways will be shown here, remember to only use */
/* one of these for you actual implementation */

#ifdef MPW3.2
        /* using MPW 3.2, use the glue */
        StdFilterProc(currentDialog,theEventIn,theDialogItem); /* MPW 3.2 glue routine */
#else
        /* not using MPW 3.2, get and call the standard filter myself */
        myErr = GetStdFilterProc(&standardProc);    /* get the standard system dialog filter
                                                    /* address */
        /* if it was not an update, we pass control to the standard filter */
        if(!myErr)
        returnVal= ((ModalFilterProcPtr)standardProc)(currentDialog,theEventIn,theDialogItem);
#endif
        SetPort(temp);}
return(returnVal);
}
```

MPW Pascal

```
{ Before we go into a ModalDialog loop, do a little preparation }
{ This inline dispatches to the standard dialog filter for you }
PROCEDURE CallStdFilterProc(theDialog: DialogPtr; VAR event: EventRecord; VAR itemHit:
INTEGER; standardProc : ProcPtr);
INLINE $205F, $4ED0;
{ This pulls the proc pointer of the stack and jumps to the standard filter, }
{ MOVE.L (SP)+,A0 }
{  JMP (A0)       }

myDialogPtr := GetNewDialog(kMyDialogID, NIL, WindowPtr(-1));
myErr :=  SetDialogDefaultItem(myDialogPtr, ok); { Tell the Dialog Manager the default item }
myErr := SetDialogCancelItem(myDialogPtr, cancel); { Tell  Dialog Manager the cancel item }
myErr :=  SetDialogTracksCursor(myDialogPtr, TRUE); { We have an edit item in our dialog,}
                                           {  so tell the Dialog Manager to change the }
                                           { cursor  to an I-Beam when it's over  edit line }
REPEAT
  ModalDialog(@MyFilter, hitItem);
UNTIL ((hitItem = ok) OR (hitItem = cancel));
```

Your filter for System 7 will look something like this:

```
FUNCTION MyFilter(currentDialog: DialogPtr; VAR theEventIn: EventRecord; VAR theItem:
INTEGER): BOOLEAN;

VAR
  savePort    : GrafPort;

BEGIN
  { if the update is for the dialog box, ignore it since the regular ModalDialog
  { function will redraw it as necessary }
  IF (theEventIn.what = updateEvt AND theEventIn.message <> currentDialog)
      MyFilter := MyDrawProc(currentDialog)
  ELSE
    BEGIN
      GetPort(savePort);            { save the current port }
      SetPort(currentDialog);       { set to the dialog, this is necessary to }
                                    { track the edit line cursor change correctly }
{ NOTE: If you are using  MPW 3.2, there is a glue routine in the Interface.o library }
{ that will take care of the  details of getting and dispatching to the standard filter }
{ for you.  If you are not using MPW 3.2, you will have to call the standard }
{  filter procedure yourself.  Both ways will be shown here, remember to only use }
{ one of these for you actual implementation }


{$IFC MPW3.2 }
       { using MPW 3.2, use the glue }
       StdFilterProccurrentDialog, theEventIn, theItem); { MPW 3.2 glue routine }
{$ELSEC}
       { not using MPW 3.2, get and call the standard filter myself }
      myErr := GetStdFilterProc(gStandardProc); {get the current system standard filter and }
                                    { store it in a global, so our assembly glue can use it }
       { if it was not an update, pass control to the assembly glue that will call the }
       { standard filter }
      IF myErr = noErr THEN
          MyFilter := CallStdFilterProc(currentDialog, theEventIn, theItem,gStandardProc);
{$ENDC}
      SetPort(savePort);            { restore the saved port }
    END;
END;
```

The System 6 Way

Of course, under pre-System 7 applications you can't use the new calls, so you have to do it yourself. Here's a sample System 6.0.x filter proc that does roughly the same thing. Of course, you can't call the new Dialog Manager routines under System 6.

MPW C

```
/* Pre-system 7 dialog filter */
pascal  Boolean  MyFilter(DialogPtr  currentDialog,  EventRecord  *theEventIn,  short
*theDialogItem)
{ /* declared as 'pascal' since it's called by the toolbox */
#define kMyButtonDelay 8
Boolean returnVal = false;
long waitTicks;
short itemKind;                              /* some temporary variables for  GetDItem use */
Handle itemHandle;
Rect itemRect;
if(theEventIn->what == updateEvt &&  theEventIn->message != myDialogPtr ) {
                                             /* myDialogPtr is defined where you created the dialog */
                                               /* if the update is for the dialog box, ignore */
                                             /* it since the regular ModalDialog function */
                                               /* will  redraw it as necessary */
 returnVal = MyDrawProc(theEventIn->message);    /* go to my drawing routine */
  } else {
```

```
                                          /* it wasn't an update, see if it was a */
                                          /* keystroke */
                                             /*  Check for the return or enter key, */
                                          /* and alias that as item 1.  */
                                             /* I also included a check here for the escape */
                                          /* key aliasing as item 2, you may not */
                                            /* want to use that */
   if ((theEventIn->what == keyDown) || (theEventIn->what == autoKey)){
   /* it was a key */
   switch (theEventIn->message & charCodeMask) {
     case kReturnKey:
     case kEnterKey:
     *theDialogItem = ok;                /* change whatever the current item is to */
                                         /* the OK item ok is #defined in Dialogs.h as 1*/
                                            /* now we need to invert the button so the */
                                         /* user gets the right feedback */
     GetDItem(currentDialog,ok,&itemKind,&itemHandle,&itemRect);
     HiliteControl((ControlHandle)itemHandle, inButton); /* invert the button */
     Delay(kMyButtonDelay , &waitTicks);   /* wait about 8 ticks so they can see it */
     HiliteControl((ControlHandle)itemHandle, false);  /* and back to normal */

     returnVal = true;                   /* tell the Dialog Manager we handled this event */
     break;
                                           /*This filters the escape key  the same as item 2 */
                                          /* (the cancel button,usually ) */

     case kEscKey:
     *theDialogItem = cancel;           /* cancel is #defined in Dialogs.h as 2 */
     GetDItem(currentDialog,cancel,&itemKind,&itemHandle,&itemRect);
      HiliteControl((ControlHandle)itemHandle, inButton);
      Delay(kMyButtonDelay , &waitTicks);/* wait about 8 ticks so they can see it */
      HiliteControl((ControlHandle)itemHandle, false);
     returnVal = true;  /* tell the Dialog Manager we handled this event */
     break;
     }
   }
  }
return(returnVal);
}
```

MPW Pascal

```
{ Your filter for pre-System 7 will look something like this: }

FUNCTION MyFilter(currentDialog: DialogPtr; VAR theEventIn: EventRecord; VAR theItem:
INTEGER): BOOLEAN;
CONST
kMyButtonDelay = 8;
VAR
  itemKind        : INTEGER;
  itemHandle      : Handle;
  itemRect        : Rect;
  savePort        : GrafPtr;
  waitTicks       : LONGINT;

BEGIN
  { if the update is for the dialog box, ignore it since the regular ModalDialog
  { function will redraw it as necessary }
  IF (theEventIn.what = updateEvt AND theEventIn.message <> currentDialog)
    MyFilter := MyDrawProc(theEventIn.message)
  ELSE { it wasn't an update, see if it was a keystroke }
    BEGIN
    { Check for the return or enter key, and alias that as item "ok".  }
    { I also included a check here for the escape key aliasing as item "cancel", }
    { you may not want to use that }
      IF ((theEventIn.what = keyDown) OR (theEventIn.what = autoKey))
        BEGIN { it was a key }
```

```
          CASE CHR(BAnd(theEventIn.message, charCodeMask)) OF

            kReturnKey, kEnterKey:
              BEGIN
                GetDItem(currentDialog, ok, itemKind, itemHandle, itemRect);
                HiliteControl(ControlHandle(itemHandle), TRUE);
                Delay(kMyButtonDelay , waitTicks);  { wait about 8 ticks so they can see it }
                HiliteControl(ControlHandle(itemHandle), FALSE);  { and back to normal }
                MyFilter := TRUE;          { tell the Dialog Manager we handled this event }
              END;

            kEscKey:
              BEGIN
                theItem := cancel;
                GetDItem(currentDialog, cancel, itemKind, itemHandle, itemRect);
                HiliteControl(ControlHandle(itemHandle), TRUE);
                Delay(kMyButtonDelay , waitTicks);  { wait about 8 ticks so they can see it }
                HiliteControl(ControlHandle(itemHandle), FALSE);  { and back to normal }
                MyFilter := TRUE;          { tell the Dialog Manager we handled this event }
              END;

          END; {CASE}
      END;
    END;
END;
```

Conclusion

Neverending updates are not a new problem, MultiFinder just makes it imperative that you do something about it. There isn't much extra work involved, just add a simple filter to all your dialogs and alerts, and put a flag to your drawing proc in your window structure.

The results will allow the system to continue to run smoothly, and as an added benefit your users will always see your application windows the way they should be, instead of windows with chunks bitten out of them.

Also, using the new Dialog Manager calls (even when you're not using a filter) allow you to present a consistent user interface across the whole system, a goal we're all striving for.

Further Reference:

- *Inside Macintosh*, Volume I, Window Manager, Dialog Manager, Event Manager

#305: PBShare, PBUnshare, and PBGetUGEntry

Written by: Jim Luther

October 1991

This Technical Note documents three new File Manager routines available on shared local volumes. The Pascal glue code, C glue code, and the assembler equates and macros for the calls are included in this note.

Three new File Manager routines, `PBShare`, `PBUnshare` and `PBGetUGEntry` are available on local volumes that have File Sharing enabled. These three routines are necessary to implement a "Sharing" dialog used to make a volume or directory a "share point" on the network and to set the Owner and User/Group of a shared folder. (For a description of share points, see Macintosh Technical Note #301.) The `PBShare` routine makes a volume or folder a share point. The `PBUnshare` routine undoes the effect of `PBShare`; it makes a share point unavailable on the network. The `PBGetUGEntry` routine lets you access the list of User and Group names and IDs on the local file server.

File Sharing should be on and the volume should be sharable before you call these three routines. You can check to see if File Sharing is turned on and that the local volume is sharable by calling the `PBHGetVolParms` routine and checking the `bHasPersonalAccessPrivileges` (local File Sharing is enabled) bit returned in the `vMAttrib` field of the `GetVolParmsInfoBuffer`. File Sharing is turned on if local File Sharing is enabled on any mounted volume. A portion of a volume can be shared only if local File Sharing is enabled on that volume. The following two functions can be used for these checks:

```
FUNCTION VolIsSharable (vRefNum: Integer): Boolean;
{See if local File Sharing is enabled on the volume specified by vRefNum}
  VAR
    pb: HParamBlockRec;
    infoBuffer: GetVolParmsInfoBuffer;
    err: OSErr;
BEGIN
  WITH pb DO
    BEGIN
      ioNamePtr := NIL;
      ioVRefNum := vRefNum;
      ioBuffer := @infoBuffer;
      ioReqCount := SizeOf(infoBuffer);
    END;
  err := PBHGetVolParmsSync(@pb);
  IF err = noErr THEN
    IF BTst(infoBuffer.vMAttrib, bHasPersonalAccessPrivileges) THEN
      VolIsSharable := TRUE
    ELSE
      VolIsSharable := FALSE
  ELSE
    VolIsSharable := FALSE;
END;
```

```
FUNCTION SharingIsOn: Boolean;
{See if File Sharing is turned on by seeing if any volume has}
{local File Sharing enabled}

  VAR
    pb: HParamBlockRec;
    err: OSErr;
    volIndex: Integer;
    sharing: Boolean;

BEGIN
  sharing := FALSE; {assume File Sharing is off}
  volIndex := 1;
  REPEAT
    WITH pb DO
      BEGIN
        ioNamePtr := NIL;
        ioVolIndex := volIndex;
      END;
    err := PBHGetVInfoSync(@pb);
    IF err = noErr THEN
      sharing := VolIsSharable(pb.ioVRefNum);
    volIndex := volIndex + 1;
  UNTIL (err <> noErr) OR sharing; {stop if error or if a volume has}
                                   {local File Sharing enabled}
  SharingIsOn := sharing;
END;
```

The Routines

Assembly-Language Note: These routines are called through the `_HFSDispatch` macro with register `A0` pointing to the parameter block and register `D0` containing a routine selector. When your completion routine is called, register `A0` points to the parameter block of the asynchronous call and register `D0` contains the result code. See *Inside Macintosh* Volume IV, pages IV-115 through IV-119, for detailed information.

PBShare

```
FUNCTION PBShare (paramblock: HParmBlkPtr; async: Boolean) :
     OSErr;
```

Trap Macro	`_Share`
Routine selector	$42

Parameter Block

→	12	`ioCompletion`	long	pointer to completion routine
←	16	`ioResult`	word	result code
→	18	`ioNamePtr`	long	pointer to directory name
→	22	`ioVRefNum`	word	volume specification
→	48	`ioDirID`	long	parent directory ID

`PBShare` makes the directory pointed to by the `ioNamePtr/ioDirID` pair on the volume specified by `ioVRefNum` a share point.

Field descriptions

`ioCompletion` Longword input pointer: A pointer to the completion routine.

`ioResult` Word result value: Result code.

`ioNamePtr` Longword input pointer: Points to the directory name, or NIL if `ioDirID` is the directory ID.

`ioVRefNum` Word input value: The volume specification (volume reference number, working directory reference number, drive number, or 0 for default volume).

`ioDirID` Longword input value: The directory or parent directory specification.

Result codes

`noErr`	0	No error
`tmfoErr`	-42	Too many share points
`fnfErr`	-43	File not found
`dupFNErr`	-48	There is already a share point with this name
`paramErr`	-50	This function is not supported
`dirNFErr`	-120	Directory not found
`afpAccessDenied`	-5000	This folder cannot be shared
`afpObjectTypeErr`	-5025	Object was a file, not a directory
`afpContainsSharedErr`	-5033	The directory contains a share point
`afpInsideSharedErr`	-5043	The directory is inside a shared directory

Pascal glue code for `PBShare`:

```
FUNCTION PBShare (paramBlock: HParmBlkPtr; async: BOOLEAN): OSErr;
  INLINE $101F,  { MOVE.B     (A7)+,D0    }
         $205F,  { MOVEA.L    (A7)+,A0    }
         $6606,  { BNE.S      *+$0008     }
         $7042,  { MOVEQ      #$42,D0     }
         $A260,  { _FSDispatch,Immed      }
         $6004,  { BRA.S      *+$0006     }
         $7042,  { MOVEQ      #$42,D0     }
         $A660,  { _FSDispatch,Sys,Immed  }
         $3E80;  { MOVE.W     D0,(A7)     }

FUNCTION PBShareSync (paramBlock: HParmBlkPtr): OSErr;
  INLINE $205F,  { MOVEA.L    (A7)+,A0    }
         $7042,  { MOVEQ      #$42,D0     }
         $A260,  { _FSDispatch,Immed      }
         $3E80;  { MOVE.W     D0,(A7)     }

FUNCTION PBShareAsync (paramBlock: HParmBlkPtr): OSErr;
  INLINE $205F,  { MOVEA.L    (A7)+,A0    }
         $7042,  { MOVEQ      #$42,D0     }
         $A660,  { _FSDispatch,Sys,Immed  }
         $3E80;  { MOVE.W     D0,(A7)     }
```

MPW C v3.1 glue code for `PBShare`:

```
pascal OSErr PBShare (HParmBlkPtr paramBlock, Boolean async)
  = {0x101F,   /* MOVE.B      (A7)+,D0    */
     0x205F,   /* MOVEA.L     (A7)+,A0    */
     0x6606,   /* BNE.S       *+$0008     */
     0x7042,   /* MOVEQ       #$42,D0     */
     0xA260,   /* _FSDispatch,Immed       */
     0x6004,   /* BRA.S       *+$0006     */
     0x7042,   /* MOVEQ       #$42,D0     */
     0xA660,   /* _FSDispatch,Sys,Immed */
     0x3E80};  /* MOVE.W      D0,(A7)     */

pascal OSErr PBShareSync (HParmBlkPtr paramBlock)
  = {0x205F,   /* MOVEA.L     (A7)+,A0    */
     0x7042,   /* MOVEQ       #$42,D0     */
     0xA260,   /* _FSDispatch,Immed       */
     0x3E80};  /* MOVE.W      D0,(A7)     */

pascal OSErr PBShareAsync (HParmBlkPtr paramBlock)
  = {0x205F,   /* MOVEA.L     (A7)+,A0    */
     0x7042,   /* MOVEQ       #$42,D0     */
     0xA660,   /* _FSDispatch,Sys,Immed */
     0x3E80};  /* MOVE.W      D0,(A7)     */
```

MPW C v3.2 glue code for `PBShare`:

```
pascal OSErr PBShare (HParmBlkPtr paramBlock, Boolean async)
  = {0x101F,   /* MOVE.B      (A7)+,D0    */
     0x205F,   /* MOVEA.L     (A7)+,A0    */
     0x6606,   /* BNE.S       *+$0008     */
     0x7042,   /* MOVEQ       #$42,D0     */
     0xA260,   /* _FSDispatch,Immed       */
     0x6004,   /* BRA.S       *+$0006     */
     0x7042,   /* MOVEQ       #$42,D0     */
     0xA660,   /* _FSDispatch,Sys,Immed */
     0x3E80};  /* MOVE.W      D0,(A7)     */

#pragma parameter __D0 PBShareSync(__A0)
pascal OSErr PBShareSync (HParmBlkPtr paramBlock)
  = {0x7042,   /* MOVEQ       #$42,D0     */
     0xA260};  /* _FSDispatch,Immed       */

#pragma parameter __D0 PBShareAsync(__A0)
pascal OSErr PBShareAsync (HParmBlkPtr paramBlock)
  = {0x7042,   /* MOVEQ       #$42,D0     */
     0xA660};  /* _FSDispatch,Sys,Immed */
```

Assembler equate and macro for `_Share`:

```
selectShare         EQU   $42

                    macro
                    _Share &async1,&async2
                          DoHFSDispatch selectShare,&async1,&async2
                    endm
```

PBUnshare

```
FUNCTION PBUnshare (paramblock: HParmBlkPtr; async: Boolean) :
    OSErr;
```

Trap Macro _Unshare
Routine selector $43

Parameter Block

→	12	ioCompletion	long	pointer to completion routine
←	16	ioResult	word	result code
→	18	ioNamePtr	long	pointer to directory name
→	22	ioVRefNum	word	volume specification
→	48	ioDirID	long	parent directory ID

PBUnshare makes the share point pointed to by the ioNamePtr/ioDirID pair on the volume specified by ioVRefNum unavailable on the network; it undoes the effect of PBShare.

Field descriptions

ioCompletion Longword input pointer: A pointer to the completion routine.

ioResult Word result value: Result code.

ioNamePtr Longword input pointer: Points to the directory name, or NIL if ioDirID is the directory ID.

ioVRefNum Word input value: The volume specification (volume reference number, working directory reference number, drive number, or 0 for default volume).

ioDirID Longword input value: The directory or parent directory specification.

Result codes

noErr	0	No error
fnfErr	-43	File not found
dirNFErr	-120	Directory not found
afpObjectTypeErr	-5025	Object was a file, not a directory, or this directory is not a share point

Pascal glue code for PBUnshare:

```
FUNCTION PBUnshare (paramBlock: HParmBlkPtr; async: BOOLEAN): OSErr;
  INLINE $101F,  { MOVE.B     (A7)+,D0    }
         $205F,  { MOVEA.L    (A7)+,A0    }
         $6606,  { BNE.S      *+$0008     }
         $7043,  { MOVEQ      #$43,D0     }
         $A260,  { _FSDispatch,Immed      }
         $6004,  { BRA.S      *+$0006     }
         $7043,  { MOVEQ      #$43,D0     }
         $A660,  { _FSDispatch,Sys,Immed  }
         $3E80;  { MOVE.W     D0,(A7)     }
```

```
FUNCTION PBUnshareSync (paramBlock: HParmBlkPtr): OSErr;
  INLINE $205F,  { MOVEA.L    (A7)+,A0    }
         $7043,  { MOVEQ      #$43,D0     }
         $A260,  { _FSDispatch,Immed      }
         $3E80;  { MOVE.W     D0,(A7)     }

FUNCTION PBUnshareAsync (paramBlock: HParmBlkPtr): OSErr;
  INLINE $205F,  { MOVEA.L    (A7)+,A0    }
         $7043,  { MOVEQ      #$43,D0     }
         $A660,  { _FSDispatch,Sys,Immed }
         $3E80;  { MOVE.W     D0,(A7)     }
```

MPW C v3.1 glue code for PBUnshare:

```
pascal OSErr PBUnshare (HParmBlkPtr paramBlock, Boolean async)
  = {0x101F,    /* MOVE.B      (A7)+,D0    */
     0x205F,    /* MOVEA.L     (A7)+,A0    */
     0x6606,    /* BNE.S       *+$0008     */
     0x7043,    /* MOVEQ       #$43,D0     */
     0xA260,    /* _FSDispatch,Immed       */
     0x6004,    /* BRA.S       *+$0006     */
     0x7043,    /* MOVEQ       #$43,D0     */
     0xA660,    /* _FSDispatch,Sys,Immed */
     0x3E80};   /* MOVE.W      D0,(A7)     */

pascal OSErr PBUnshareSync (HParmBlkPtr paramBlock)
  = {0x205F,    /* MOVEA.L     (A7)+,A0    */
     0x7043,    /* MOVEQ       #$43,D0     */
     0xA260,    /* _FSDispatch,Immed       */
     0x3E80};   /* MOVE.W      D0,(A7)     */

pascal OSErr PBUnshareAsync (HParmBlkPtr paramBlock)
  = {0x205F,    /* MOVEA.L     (A7)+,A0    */
     0x7043,    /* MOVEQ       #$43,D0     */
     0xA660,    /* _FSDispatch,Sys,Immed */
     0x3E80};   /* MOVE.W      D0,(A7)     */
```

MPW C v3.2 glue code for PBUnshare:

```
pascal OSErr PBUnshare (HParmBlkPtr paramBlock, Boolean async)
  = {0x101F,    /* MOVE.B      (A7)+,D0    */
     0x205F,    /* MOVEA.L     (A7)+,A0    */
     0x6606,    /* BNE.S       *+$0008     */
     0x7043,    /* MOVEQ       #$43,D0     */
     0xA260,    /* _FSDispatch,Immed       */
     0x6004,    /* BRA.S       *+$0006     */
     0x7043,    /* MOVEQ       #$43,D0     */
     0xA660,    /* _FSDispatch,Sys,Immed */
     0x3E80};   /* MOVE.W      D0,(A7)     */

#pragma parameter __D0 PBUnshareSync(__A0)
pascal OSErr PBUnshareSync (HParmBlkPtr paramBlock)
  = {0x7043,    /* MOVEQ       #$43,D0     */
     0xA260};   /* _FSDispatch,Immed       */

#pragma parameter __D0 PBUnshareAsync(__A0)
pascal OSErr PBUnshareAsync (HParmBlkPtr paramBlock)
  = {0x7043,    /* MOVEQ       #$43,D0     */
     0xA660};   /* _FSDispatch,Sys,Immed */
```

Assembler equate and macro for `_Unshare`:

```
selectUnshare        EQU   $43

                    macro
                    _Unshare &async1,&async2
                          DoHFSDispatch selectUnshare,&async1,&async2
                    endm
```

PBGetUGEntry

```
FUNCTION PBGetUGEntry (paramblock: HParmBlkPtr; async: Boolean) :
    OSErr;
```

Trap Macro `_GetUGEntry`
Routine selector $44

Parameter Block

→	12	ioCompletion	long	pointer to completion routine
←	16	ioResult	word	result code
→	26	ioObjType	word	object type function code
→	28	ioObjNamePtr	long	ptr to returned user/group name
↔	32	ioObjID	long	user/group ID

`PBGetUGEntry` asks the local file server for the next user or group in its list. `PBGetUGEntry` returns the user or group name and the user or group ID.

Field descriptions

`ioCompletion` Longword input pointer: A pointer to the completion routine.

`ioResult` Word result value: Result code.

`ioObjType` Word input value: Determines the type of object to be returned, as follows:
$0001 return next user
$0002 return next group
$0003 return next user or next group

`ioObjNamePtr` Longword input pointer: Points to a result buffer where the user or group name is to be returned. If the pointer is NIL, then no name is returned. The name is returned as a Pascal string with a maximum size of 31 characters (Str31).

`ioObjID` Longword input/result value: The server will return the first user or group whose name is alphabetically next from the user specified by `ioObjID`. Setting `ioObjID` to 0 will return the first user or group. On return, `ioObjID` will be the user or group's ID.

You can enumerate the user or group list in alphabetical order by calling this routine again and again without changing the parameter block until the result code `fnfErr` is returned.

Result codes

`noErr`	0	No error
`paramErr`	-50	The `ioObjID` is negative or this function is not supported
`fnfErr`	-43	There are no more users or groups to return

Pascal glue code for `PBGetUGEntry`:

```
FUNCTION PBGetUGEntry (paramBlock: HParmBlkPtr; async: BOOLEAN): OSErr;
  INLINE $101F,   { MOVE.B     (A7)+,D0    }
         $205F,   { MOVEA.L    (A7)+,A0    }
         $6606,   { BNE.S      *+$0008     }
         $7044,   { MOVEQ      #$44,D0     }
         $A260,   { _FSDispatch,Immed      }
         $6004,   { BRA.S      *+$0006     }
         $7044,   { MOVEQ      #$44,D0     }
         $A660,   { _FSDispatch,Sys,Immed }
         $3E80;   { MOVE.W     D0,(A7)     }

FUNCTION PBGetUGEntrySync (paramBlock: HParmBlkPtr): OSErr;
  INLINE $205F,   { MOVEA.L    (A7)+,A0    }
         $7044,   { MOVEQ      #$44,D0     }
         $A260,   { _FSDispatch,Immed      }
         $3E80;   { MOVE.W     D0,(A7)     }

FUNCTION PBGetUGEntryAsync (paramBlock: HParmBlkPtr): OSErr;
  INLINE $205F,   { MOVEA.L    (A7)+,A0    }
         $7044,   { MOVEQ      #$44,D0     }
         $A660,   { _FSDispatch,Sys,Immed }
         $3E80;   { MOVE.W     D0,(A7)     }
```

MPW C v3.1 glue code for `PBGetUGEntry`:

```
pascal OSErr PBGetUGEntry (HParmBlkPtr paramBlock, Boolean async)
  = {0x101F,   /* MOVE.B     (A7)+,D0    */
     0x205F,   /* MOVEA.L    (A7)+,A0    */
     0x6606,   /* BNE.S      *+$0008     */
     0x7044,   /* MOVEQ      #$44,D0     */
     0xA260,   /* _FSDispatch,Immed      */
     0x6004,   /* BRA.S      *+$0006     */
     0x7044,   /* MOVEQ      #$44,D0     */
     0xA660,   /* _FSDispatch,Sys,Immed */
     0x3E80};  /* MOVE.W     D0,(A7)     */

pascal OSErr PBGetUGEntrySync (HParmBlkPtr paramBlock)
  = {0x205F,   /* MOVEA.L    (A7)+,A0    */
     0x7044,   /* MOVEQ      #$44,D0     */
     0xA260,   /* _FSDispatch,Immed      */
     0x3E80};  /* MOVE.W     D0,(A7)     */

pascal OSErr PBGetUGEntryAsync (HParmBlkPtr paramBlock)
  = {0x205F,   /* MOVEA.L    (A7)+,A0    */
     0x7044,   /* MOVEQ      #$44,D0     */
     0xA660,   /* _FSDispatch,Sys,Immed */
     0x3E80};  /* MOVE.W     D0,(A7)     */
```

MPW C v3.2 glue code for PBGetUGEntry:

```
pascal OSErr PBGetUGEntry (HParmBlkPtr paramBlock, Boolean async)
  = {0x101F,   /* MOVE.B     (A7)+,D0   */
     0x205F,   /* MOVEA.L    (A7)+,A0   */
     0x6606,   /* BNE.S      *+$0008    */
     0x7044,   /* MOVEQ      #$44,D0    */
     0xA260,   /* _FSDispatch,Immed     */
     0x6004,   /* BRA.S      *+$0006    */
     0x7044,   /* MOVEQ      #$44,D0    */
     0xA660,   /* _FSDispatch,Sys,Immed */
     0x3E80};  /* MOVE.W     D0,(A7)    */

#pragma parameter __D0 PBGetUGEntrySync(__A0)
pascal OSErr PBGetUGEntrySync (HParmBlkPtr paramBlock)
  = {0x7044,   /* MOVEQ      #$44,D0    */
     0xA260};  /* _FSDispatch,Immed     */

#pragma parameter __D0 PBGetUGEntryAsync(__A0)
pascal OSErr PBGetUGEntryAsync (HParmBlkPtr paramBlock)
  = {0x7044,   /* MOVEQ      #$44,D0    */
     0xA660};  /* _FSDispatch,Sys,Immed */
```

Assembler equate and macro for _GetUGEntry:

```
selectGetUGEntry    EQU   $44

                    macro
                    _GetUGEntry &async1,&async2
                          DoHFSDispatch selectGetUGEntry,&async1,&async2
                    endm
```

Further Reference:

- *Inside Macintosh*, Volume IV, The File Manager
- *Inside Macintosh*, Volume V, File Manager Extensions In a Shared Environment
- *Inside Macintosh*, Volume VI, The File Manager
- *Inside AppleTalk*, AppleTalk Filing Protocol
- Technical Note #301, File Sharing and Shared Folders

Macintosh
Technical Notes

Developer Technical Support

#306: Drawing Icons the System 7 Way

Revised by: Jim Mensch — May 1992
Written by: Jim Mensch and David Collins — October 1991

This Technical Note describes how to utilize the built-in System 7 icon drawing utility. Use this information to better conform to the System 7.0 visual human interface.

Introduction

With the introduction of System 7.0 for the Macintosh, Apple has defined a new look and feel for many screen elements that better utilize color. Among other elements, the icons drawn by the Finder and other system components have been redefined. While Apple has documented how to create this new look for most elements, Apple has not documented how to draw the icons the way the Finder does in System 7. This Technical Note discusses all the icon toolkits that the Finder uses to draw and manipulate the icons on the screen. Two of the calls, PlotIconID and PlotCIconHandle, are the ones you will probably use the most since they deal with simply drawing single icons to the screen. Other places in the Toolbox require that an icon family handle be passed to them to allow the drawing of color icons. The toolkit provides calls that allow you to create, draw, and manipulate these handles. What follows is a description of the new icon data structures and the calls in the icon toolkit.

The New 'ic' Type Resources

PlotIconID and PlotCIconHandle allow the use of standard CIcons as documented in *Inside Macintosh* Volume V and the use of a new set of icon resources utilized by the PlotIconID call. This new icon type is actually not a single resource but a collection of many different icons into a family. Each member of the family has a consistent resource ID and a resource type indicating what type of icon data is stored in that particular resource. Currently Apple has defined three sizes of icons and three different bit depths for each size. The sizes are large (32x32 pixels), small (16x16 pixels), and mini (12x12 pixels), and the bit depths are 1, 4, and 8. The actual resource types are defined as follows:

```
        Large1BitMask       =       'ICN#';
        Large4BitData       =       'icl4';
        Large8BitData       =       'icl8';
        Small1BitMask       =       'ics#';
        Small4BitData       =       'ics4';
        Small8BitData       =       'ics8';
        Mini1BitMask        =       'sicn';
        Mini4BitData        =       'icm4';
        Mini8BitData        =       'icm8';
```

The 1-bit-per-pixel member of each size also contains the mask data for all icons of that size (yes, this means that all your icons of a certain size must have the same mask). A 1-bit-per-pixel member must exist for each icon size you want PlotIconID to use. The icon size to use is determined by the size of the destination rectangle. If the destination rect is greater than 16 pixels on a side then the large icon will be used. If the destination rect is 13–16 pixels on both sides the small icon will be used. If the destination is 12 or less on each side the mini icon will be used. The bit depth is determined by the device of the grafPort you plot into at drawing time. Be sure that you always create a color grafPort any time you want to use color icons.

Icon Families (or Suites and Caches As the Tool Set Refers to Them)

An icon family is simply a collection of icon handles that contain up to one image of each bit depth and size for a given icon. By using families, you remove the need to determine which size or depth of icon to use when drawing into a given rectangle. Several system routines can take an icon family handle when an icon is requested (the Notification Manager for example) so that the proper color icons can be used if available. An icon family can be fully populated (every possible size or depth available) or it can have only those icons that exist or are needed. In the case of a sparsely populated icon family, if the proper icon is not available the icon tool set will pick an appropriate substitute that will produce the best results.

An icon cache is a family that also has a ProcPtr and a refCon. The main difference between a cache and a family is that the elements of the cache's array are sparsely populated. When using an icon cache, the system either will use the entry in the icon family portion of the cache, or if the desired element is empty, it will call your procPtr and request the data for the icon. The Proc has the following calling convention:

```
Function GetAnIcon(theType:ResType; yourDataPtr:Ptr):Handle;
```

This function should return either the icon data to be used to draw or NIL to signify that this entry in the icon cache does not exist. Icon caches can be used with all icon family calls and have a few extra calls used to manipulate them.

Now that we know about the different data types let's examine how to manipulate the drawing:

Drawing Modes or Transforms

In addition to being drawn in various sizes and bit depths, icons can be drawn with different "Modes" or transforms. Transforms are analogous to certain Finder states for the icons. For example, the transform that you would use to show an icon of a disk that has been ejected is ttOffline. Here is a list of the current transforms that are available:

```
{ IconTransformType values }
ttNone                     =      $0;
ttDisabled                 =      $1;
ttOffline                  =      $2;
ttOpen                     =      $3;
ttSelected                 =      $4000;
ttSelectedDisabled         =      (ttSelected + ttDisabled);
ttSelectedOffline          =      (ttSelected + ttOffline);
ttSelectedOpen             =      (ttSelected + ttOpen);
```

The actual appearance of the icon drawn by each transform type may vary with future system software, so you should always try to use the transform that best fits the state it represents in your application. In this way you will be consistent with any possible future changes to the look and feel of regular system icons. Note that the ttSelected transform can be added to any of the other transform types. Additional transform types exist for displaying the icon of a file inside your application that use the Finder label colors to color the icon. To determine the proper label for a file's icon, you can check bits 1–3 of the fdFlags field in the file's Finder info (See the File Manager chapter in *Inside Macintosh* Volume IV for more information). These bits contain a number from 0 to 7. Simply add the corresponding ttLabel value to the transform that you give the call. The label values are defined like this:

```
ttLabel0                    =       $0000;
ttLabel1                    =       $0100;
ttLabel2                    =       $0200;
ttLabel3                    =       $0300;
ttLabel4                    =       $0400;
ttLabel5                    =       $0500;
ttLabel6                    =       $0600;
ttLabel7                    =       $0700;
```

Alignment

Most icons do not fully fill their rectangle, and it is sometimes necessary to draw an icon relative to other data (like menu text). In these instances it would be nice to be able to have the icon move in its rectangle so that it will be at a predictable location in the destination rectangle. Therefore, when drawing an icon you can pass one of these standard alignments in the alignment parameter or you can add a vertical alignment to a horizontal alignment to create a composite alignment value.

```
atNone                      =       $0;
atVerticalCenter            =       $1;
atTop                       =       $2;
atBottom                    =       $3;
atHorizontalCenter          =       $4;
atLeft                      =       $8;
atRight                     =       $C;
```

And Now (Drum Roll Please) the Calls and What to Pass

Now that we have defined every major data type we can think of, here is the real meat of this Tech Note: the actual calls themselves. I am providing only the Pascal and C interfaces here since they do not appear in your current MPW Interfaces folder. Disk copies of these can be found on AppleLink and on current and future versions of the *Developer CD Series* disc.

Icon Family Calls

```
Function   NewIconSuite(  var  theSuite:Handle):OSErr;
```

This call returns an empty icon family handle with all members set to NIL.

```
Function   AddIconToSuite(  theIconData:  Handle;  theSuite:Handle;
                            theType:ResType):OSErr;
```

This call will add the data in theIconData into the suite at the location reserved for theType of icon data. This call will replace any old data in that slot without disposing of it, so you may want to call GetIconFromSuite to obtain the old handle (if any) to dispose. This call will be used most often with the NewIconSuite call to fill the empty family after it's created.

```
Function   GetIconFromSuite(  Var  theIconData:Handle;  theSuite:Handle;
                                theType:ResType):OSErr;
```

This call will return a handle to the pixel data of the family member of theSuite specified by theType. If you intend to dispose of this handle, be sure to call AddIconToSuite with a NIL handle to zero out the family entry.

```
Function   ForEachIconDo( theSuite:  Handle;  theSelector:Longint;
                          actionProc:procPtr;  yourData:  UNIV  ptr):OSErr;
```

This routine will call your actionProc for each icon in the family specified by theSelector and theSuite. TheSelector is a bit level flag that specifies which family members to operate on; they can be added together to create composite selectors that work on several different family members. The values for theSelector are as follows:

```
{ IconSelectorValue masks }
svLarge1Bit                 =      $00000001;
svLarge4Bit                 =      $00000002;
svLarge8Bit                 =      $00000004;
svSmall1Bit                 =      $00000100;
svSmall4Bit                 =      $00000200;
svSmall8Bit                 =      $00000400;
svMini1Bit                  =      $00010000;
svMini4Bit                  =      $00020000;
svMini8Bit                  =      $00040000;
svAllLargeData              =      $000000ff;
svAllSmallData              =      $0000ff00;
svAllMiniData               =      $00ff0000;
svAll1BitData               =      (svLarge1Bit + svSmall1Bit + svMini1Bit);
svAll4BitData               =      (svLarge4Bit + svSmall4Bit + svMini4Bit);
svAll8BitData               =      (svLarge8Bit + svSmall8Bit + svMini8Bit);
svAllAvailableData          =      $ffffffff;
```

The action procedure that gets called for each icon type selected for the family is a Pascal type function with the following interface:

```
Function ActionProc(var theIconData:Handle; theType:ResType; yourDataPtr: UNIV ptr):OSErr;
```

theIconData is passed by reference here so that your routine can modify the contents of the suite directly. yourDataPtr is the value passed when you called forEachIconDo; it allows you to easily communicate with your application. The action procedure returns an OSErr; if any value other than noErr is returned, forEachIconDo will stop processing immediately and return the error passed. (Note: This implies that the icons selected may only be partially operated on.) There is no guaranteed order in which the icons get operated on.

```
Function   GetIconSuite(  Var  theSuite:  Handle;  theID:Integer;
                          theSelector:Longint):OSErr;
```

GetIconSuite will create a new icon family and then fill it with the icons with the passed ID, of the indicated types in theSelector, from the current resource chain. This is the call you will probably use most often to create an icon family. Note: If you SetResLoad(False) before making this call, the suite will be filled with unloaded resource handles.

```
Function   PlotIconSuite(   theRect:Rect;   alignment:Integer;   transform:Integer;
                            theSuite:Handle):OSErr;
```

This call renders the proper icon image from the passed icon family based on the bit depth of the display you are using and the rectangle that you have passed. Alignment and transform are applied to the icon selected for drawing and then the icon is plotted into the current grafPort. PlotIconSuite chooses the appropriate icon based primarily on size; once the proper icon size is determined (based on the destination rectangle) the present member of that size with the deepest bit depth that the current device can use is selected. A size category is considered present if the black and white member (with mask) is present, ICN#, ics#, or icm#. PlotIconSuite can be used for both picture accumulation and printing.

```
Function   DisposeIconSuite(  theSuite:Handle;   disposeData:Boolean):OSErr;
```

This call disposes the icon family handle itself; in addition if disposeData is true, any of the icon data handles that do not belong to a resource fork will also be disposed.

```
Function   SetSuiteLabel(  theSuite:  Handle;  theLabel:Integer):OSErr;
```

This call allows you to specify a label that is used to draw an icon of this suite when noLabel is specified in PlotIconSuite. This is used primarily when you want to make sure that a family passed to a system routine gets drawn with the proper label. The default label can be overridden by specifying a label in PlotIconSuite.

```
Function   GetSuiteLabel(  theSuite:Handle;  var  theLabel:Integer):OSErr;
```

returns any label set with SetSuiteLabel previously.

Icon Cache Calls

In addition to the icon family calls above, icon caches have these additional calls:

```
Function   MakeIconCache(  VAR  theCache:Handle;  GetAnIcon:ProcPtr;
                           yourDataPtr:Ptr):OSErr;
```

This call creates an empty icon cache similar to NewIconSuite, and associates the additional icon loading proc and data value with the family.

```
Function   LoadIconCache(  theRect:Rect;  alignment:Integer;  transform:Integer;
                           theSuite:Handle):OSErr;
```

This call allows you to preflight the loading of certain elements of your icon cache. This is handy when you suspect that certain drawing operations will occur at a time not convenient for you to load your icon data (for example, when your resource fork might not be in open chain). LoadIconCache takes the same parameters as PlotIconSuite and uses the same criterion to select the icon to load. Be sure that the grafPort is set properly before you make this call since it is part of the criterion for determining which icon to load.

The following four calls are provided to allow you to change the dataPtr and procPtr associated with an icon cache:

```
Function   GetCacheData(  theCache:Handle;  VAR  yourDataPtr:Ptr):OSErr;
Function   SetCacheData(  theCache:Handle;    yourDataPtr:Ptr):OSErr;
Function   GetCacheProc(  theCache:Handle;  VAR  theProc:ProcPtr):OSErr;
Function   SetCacheProc(  theCache:Handle;    theProc:ProcPtr):OSErr;
```

Plotting Icons Not Part of a Suite

The following two calls are grouped because they are very similar. These routines let you simply plot an icon to the screen without having to create an icon suite. They are also good if you have a cicn instead of an icon family.

```
FUNCTION PlotIconID( TheRect: Rect;
                     Align: Integer;
                     Transform: Integer;
                     TheResID: INTEGER): OSErr;

FUNCTION PlotCIconHandle(TheRect: Rect;
                     Align: Integer;
                     transform: Integer;
                     TheCIcon: CIconHandle): OSErr;
```

TheRect is the destination rectangle to draw the indicated icon into.

Align is the alignment method to use if the icon does not exactly fit the rectangle given. Pass zero for this value. See the next version of this Tech Note for more information on alignment.

Transform indicated the desired appearance of the icon on the screen.

TheResID is the resource ID of the family of 'ic' type resources to use. If the correct bit depth or size required is not defined, the closest-fitting one will be used.

TheCIcon is a handle that you get to a standard QuickDraw color icon. Call GetCIcon to load these and do not forget to dispose of it when you are done (sometimes they can take up quite a bit of memory).

Both functions return an error code if all did not go well with the drawing, or in the case of the PlotIconID call, if the indicated icon family could not be used.

Miscellaneous Calls

```
Function   GetLabelColor(  labelNumber:Integer;   var   labelColor:RGBColor;
                           VAR   LabelString:str255):OSErr;
```

This call returns the actual color and string used in the label menu of the Finder and the label's Control Panel. This information is provided in case you wish to include the label text or color when displaying a file's icon in your application.

```
Function   IconSuiteToRgn(   theRgn:  RgnHandle;   iconRect:Rect
                             Alignment:Integer;    iconSuite:Handle):OSErr;
Function   IconIDToRgn(      theRgn:  RgnHandle;   iconRect:Rect
                             Alignment:Integer;    iconID:Integer):OSErr;
```

These routines will create a region from the mask of the icon selected by the Rect and Alignment passed. This will allow you to do accurate hit testing and outline dragging of an icon in your application. TheRgn handle must have been previously allocated before you make this call.

```
Function   RectInIconSuite( testRect:Rect;   iconRect:Rect;   Alignment:Integer;
                            IconSuite:Handle):Boolean;
Function   RectInIconID(    testRect:Rect;   iconRect:Rect;   Alignment:Integer;
                            IconID:Integer):Boolean;
Function   PtInIconSuite(   testPoint:Point;   iconRect:Rect;   Alignment:Integer;
                            IconSuite:Handle):Boolean;
Function   PtInIconID(      testPoint:Point;   iconRect:Rect;   Alignment:Integer;
                            IconID:Integer):Boolean;
```

These calls hit test the passed point or rect again the icon indicated. The iconRect, alignment, and grafPort should be the same as when the icon was drawn last. They return true if the point is in the icon mask, or if the rect intersects the icon mask.

Glue for C and Pascal

Since the standard interface files do not contain the glue for these calls, I am going to include it here since Tech Notes sometimes get distributed in electronic format and if all else fails you can copy and paste it.

```
{ Pascal Glue }
FUNCTION PlotIconID(theRect: Rect;align: Integer;transform: Integer;
       theResID: INTEGER): OSErr;   INLINE $303C, $0500, $ABC9;
FUNCTION NewIconSuite(VAR theIconSuite: Handle): OSErr;   INLINE $303C, $0207, $ABC9;
FUNCTION AddIconToSuite(theIconData: Handle;theSuite: Handle;theType: ResType): OSErr;
       INLINE $303C, $0608, $ABC9;
FUNCTION GetIconFromSuite(VAR theIconData: Handle;theSuite: Handle;theType: ResType): OSErr;
       INLINE $303C, $0609, $ABC9;
FUNCTION ForEachIconDo(theSuite: Handle;selector: Integer;action: ProcPtr;
       yourDataPtr: Ptr): OSErr;    INLINE $303C, $060A, $ABC9;
FUNCTION GetIconSuite(VAR theIconSuite: Handle;theResID: INTEGER;
       selector: Integer): OSErr;   INLINE $303C, $0501, $ABC9;
FUNCTION DisposeIconSuite(theIconSuite: Handle;disposeData: BOOLEAN): OSErr;
       INLINE $303C, $0302, $ABC9;
FUNCTION PlotIconSuite(theRect: Rect;align: Integer;transform: Integer;
       theIconSuite: Handle): OSErr;        INLINE $303C, $0603, $ABC9;
FUNCTION MakeIconCache(VAR theHandle: Handle;makeIcon: procPtr;
       yourDataPtr: UNIV Ptr): OSErr;       INLINE $303C, $0604, $ABC9;
FUNCTION LoadIconCache(theRect: Rect;align: Integer;transform: Integer;
       theIconCache: Handle): OSErr;        INLINE $303C, $0606, $ABC9;
FUNCTION GetLabel(labelNumber: INTEGER; VAR labelColor: RGBColor;
       VAR labelString: Str255): OSErr;     INLINE $303C, $050B, $ABC9;
FUNCTION PtInIconID(testPt: Point; iconRect: Rect; align: Integer;
        iconID: INTEGER): BOOLEAN;  INLINE $303C, $060D, $ABC9;
FUNCTION PtInIconSuite(testPt: Point; iconRect: Rect;align: Integer;
       theIconSuite: Handle): BOOLEAN;      INLINE $303C, $070E, $ABC9;
FUNCTION RectInIconID(testRect: Rect; iconRect: Rect;align: Integer;
       iconID: INTEGER): BOOLEAN;   INLINE $303C, $0610, $ABC9;
FUNCTION RectInIconSuite(testRect: Rect; iconRect: Rect;align: Integer;
       theIconSuite: Handle): BOOLEAN;      INLINE $303C, $0711, $ABC9;
FUNCTION IconIDToRgn(theRgn: RgnHandle; iconRect: Rect;align: Integer;
       iconID: INTEGER): OSErr;     INLINE $303C, $0913, $ABC9;
FUNCTION IconSuiteToRgn(theRgn: RgnHandle; iconRect: Rect;align: Integer;
       theIconSuite: Handle): OSErr;        INLINE $303C, $0914, $ABC9;
FUNCTION SetSuiteLabel(theSuite: Handle; theLabel: INTEGER): OSErr;
       INLINE $303C, $0316, $ABC9;
FUNCTION GetSuiteLabel(theSuite: Handle): INTEGER; INLINE $303C, $0217, $ABC9;
FUNCTION GetIconCacheData(theCache: Handle; VAR theData: Ptr): OSErr;
       INLINE $303C, $0419, $ABC9;
FUNCTION SetIconCacheData(theCache: Handle; theData: Ptr): OSErr;
       INLINE $303C, $041A, $ABC9;
FUNCTION GetIconCacheProc(theCache: Handle; VAR theProc: ProcPtr): OSErr;
       INLINE $303C, $041B, $ABC9;
FUNCTION SetIconCacheProc(theCache: Handle; theProc: procPtr): OSErr;
       INLINE $303C, $041C, $ABC9;
FUNCTION PlotCIconHandle(theRect: Rect; align: INTEGER; transform: INTEGER;
       theCIcon: CIconHandle): OSErr;       INLINE $303C, $061F, $ABC9;
```

```
/* C Glue */
pascal OSErr PlotIconID(const Rect *theRect, short align, short  transform, short theResID)
        = {0x303C, 0x0500, 0xABC9};
pascal OSErr NewIconSuite(Handle *theIconSuite) = {0x303C, 0x0207, 0xABC9};
pascal OSErr AddIconToSuite(Handle theIconData,Handle theSuite,ResType theType)= {0x303C, 0x0608, 0xABC9};
pascal OSErr GetIconFromSuite(Handle *theIconData,Handle theSuite,ResType theType)= {0x303C, 0x0609, 0xABC9};
pascal OSErr ForEachIconDo(Handle theSuite,short selector,ProcPtr action,void *yourDataPtr)
        = {0x303C, 0x080A, 0xABC9};
pascal OSErr GetIconSuite(Handle *theIconSuite,short theResID,short selector)= {0x303C, 0x0501, 0xABC9};
pascal OSErr DisposeIconSuite(Handle theIconSuite,Boolean disposeData)= {0x303C, 0x0302, 0xABC9};
pascal OSErr PlotIconSuite(const Rect *theRect,short align,short transform,Handle theIconSuite)
        = {0x303C, 0x0603, 0xABC9};
pascal OSErr MakeIconCache(Handle *theHandle,ProcPtr makeIcon,void *yourDataPtr)= {0x303C, 0x0604, 0xABC9};
pascal OSErr LoadIconCache(const Rect *theRect,short align,short transform,Handle theIconCache)
        = {0x303C, 0x0606, 0xABC9};
pascal OSErr GetLabel(short labelNumber,RGBColor *labelColor,Str255 labelString)= {0x303c, 0x050B, 0xABC9};
pascal Boolean PtInIconID(Point testPt,Rect *iconRect,short alignment,short iconID)= {0x303c, 0x060D, 0xABC9};
pascal Boolean PtInIconSuite(Point testPt,Rect *iconRect,short alignment,Handle theIconSuite)
        = {0x303c, 0x070E, 0xABC9};
pascal Boolean RectInIconID(Rect *testRect,Rect *iconRect,short alignment,short iconID)
        = {0x303c, 0x0610, 0xABC9};
pascal Boolean RectInIconSuite(Rect *testRect,Rect *iconRect,short alignment,Handle theIconSuite)
        = {0x303c, 0x0711, 0xABC9};
pascal OSErr IconIDToRgn(RgnHandle theRgn,Rect *iconRect,short alignment,short iconID)
        = {0x303c, 0x0613, 0xABC9};
pascal OSErr IconSuiteToRgn(RgnHandle theRgn,Rect *iconRect,short alignment,Handle theIconSuite)
        = {0x303c, 0x0714, 0xABC9};
pascal OSErr SetSuiteLabel(Handle theSuite, short theLabel)= {0x303C, 0x0316, 0xABC9};
pascal short GetSuiteLabel(Handle theSuite)= {0x303C, 0x0217, 0xABC9};
pascal OSErr GetIconCacheData(Handle theCache, void **theData)= {0x303C, 0x0419, 0xABC9};
pascal OSErr SetIconCacheData(Handle theCache, void *theData)= {0x303C, 0x041A, 0xABC9};
pascal OSErr GetIconCacheProc(Handle theCache, ProcPtr *theProc)= {0x303C, 0x041B, 0xABC9};
pascal OSErr SetIconCacheProc(Handle theCache, ProcPtr theProc)= {0x303C, 0x041C, 0xABC9};
pascal OSErr PlotSICNHandle(const Rect *theRect,short align,short transform,Handle theSICN)
        = {0x303C, 0x061E, 0xABC9};
pascal OSErr PlotCIconHandle(const Rect *theRect,short align,short transform,CIconHandle theCIcon)
        = {0x303C, 0x061F, 0xABC9};
pascal OSErr SetLabel(short labelNumber, const RGBColor *, ConstStr255Param)
        = {0x303C, 0x050C, 0xABC9};
```

Further Reference:

- *Inside Macintosh,* Volume V, QuickDraw chapter

Macintosh
Technical Notes

Developer Technical Support

#307: MPW C++ Pitfalls

New version by: Kent Sandvik May 1992
Original by: Kent Sandvik, Kim Coleman, and Preston Gardner January 1992

This Technical Note covers most of the common and serious subtle problems that a MPW C++ user might encounter. For more information consult the current C++ literature. This Note will be updated periodically to reflect changes in the language and the compiler. Always read the release notes included with the MPW C++ to find out the latest status for known bugs and restrictions.

Changes since January 1992: The original inline C++ Tech Note is now part of an overall MPW C++ problem Tech Note.

1. Introduction

C++, like any other computer language, has its own subtle problems, traps, and pitfalls. It is impossible to figure out all the possible pitfalls that may occur, but this Tech Note covers the most frequently asked questions about MPW C++ problems.

2. Class Protection and Inheritance

Access Control Problems

The C++ compiler assumes private access control if any of the access control keywords are omitted. For instance, in the following case the member function `Run` is declared `private`, and thus is not accessible from the outside the class:

```
class TClass
{
        TClass() {/* constructor code *}
        void Run();             // private member function
// ...
};
```

This is also true when using inheritance; if no keywords are included, the compiler assumes that the base class is inherited as a private member:

```
class TFoo : TClass
{
// ...
};
```

Always exercise special care when using inheritance, and use the keywords `private`, `protected`, or `public` to avoid unexpected problems.

Derived Classes

Every member function must function properly for the same range of arguments accepted by the base class. If not, then the derived class is not a true subtype of the base and you may encounter subtle and bizarre problems that are hard to find.

Note that if you inherit the base class as a private class, it is the same as if the class were a private member of the derived class. Thus there are few cases where one needs to import a base class as a private class.

Be especially careful to avoid changing the meaning of a base class's public interface. Any public member function of the base class should not have its semantics changed by the derived class. Fortunately the MPW C++ compiler warns you if this is done.

Here's an example: if a class `TBase` has an overloaded `operator ==` for comparison that takes a `const TBase&` as an argument, then any override of this function by a derived class `TDerived` must preserve its semantics. In particular the `TDerived` class override may not assume that the argument is of type `const TDerived&`, as that changes the meaning of the member function that is inherited from `TBase`'s public interface.

In this case it would be better to overload `operator ==` to accept an argument of type `const TDerived&`, and to reexport the inherited `operator ==`. Thus you need to overload the derived class's operator if the comparison overload will make use of a new data structure. Another solution would be to use a pointer to a function/member function to define the actual comparison routine instead of assuming there is a fixed comparison routine. Careful analysis of the use of data structures should help you avoid these problems. Bugs related to this particular problem are extremely difficult to track down, especially when the class inherits from two or more base classes, each of which defines a function with the same name but with different semantics.

Derived Classes as Variables

Any member function that accepts a reference or pointer to a class must be prepared to receive a derived class as an actual argument. Therefore the recipient function must deal with the argument through an interface that is guaranteed to be preserved in derived classes. If this is not the case the function call will fail when used with derived classes. If this is not feasible, then the class documentation should state that it cannot be used as a public base class.

Scoping Issues

The scope rules in C++ are in flux. The earlier C++ compilers did not protect the name space concerning scoping of types declared inside classes. This has changed in MPW C++ 3.2. You are now able to define `typedefs`, `enums`, and classes/structs with a class scope. However, read the release notes for your MPW C++ version for possible known bugs and limitations concerning this new feature.

3. Type Casting

The Problem: Conversion Versus Coercion Type Casting

Type casting is occasionally necessary in C and C++, but you should be aware of the consequences every time you need to use it. Casts are very uncontrolled and dangerous, and you should ask yourself if you really need to do one every time you catch yourself in the act.

There are two types of casts in C++. The first one merely changes an object from one type to another. This includes casts between the built-in arithmetic types and casts involving class objects (not pointers to classes). These are in general fairly safe, since an actual conversion is taking place.

The other type of cast involves pointers, and these casts are dangerous. This type of cast involves so-called type coercion: the bit pattern of one type is interpreted as another type. This is very unsafe, and could cause the code to die mysteriously and subtly.

Unfortunately, some C++ constructs can be interpreted as either of the two types of casts. A cast from one class pointer to another is interpreted as a conversion cast if the two types are related by type inheritance and as coercion casts if they are not. The C++ compiler does not warn you if you intend the first cast but wind up with the second. Worse, a cast between pointers to member functions may be a conversion on the class part but a coercion on the function prototype part.

Casting Cases

Some casts are always of the coercion type. For instance, casting a const (and a future volatile) pointer to one without those attributes is always a coercion. Avoid performing such casts. If you make a member function const because it does not change the object semantics, then you must cast your this pointer to non-const to make changes to the internal object state. However, this technique is not recommended; instead you should overload the function or make another design decision.

There are also casts to and from `void*`. These are dangerous. Avoid such casts, even if the `void*` is a useful construct. For instance, do not use `void*` to avoid assigning a type to a variable or parameter. Use it only for manipulation of raw storage.

Even if casts from a base class pointer to a derived class pointer are conversions, you should avoid these. First, if you accidentally specify types not related by inheritance, you will get a silent coercion. Second, this is a poor programming technique and removes vital information used for type checking. The future template support in C++ should obviate the need for most such casts.

In general the only normally acceptable cast is the conversion type. Avoid all casts involving pointers unless absolutely necessary. Note that nonpointer casts can never silently become coercions.

4. General Class Issues

Handling Failing Constructors

Constructors and destructors do not return any values, so a returned error code is not possible. There are many ways to provide error handling with constructors/destructors.

For instance, the class could have an internal field that signals whether the construction of the class succeeded, as well as a special test method or invariant method that checks whether the state of the newly created class is valid. Failing to figure out if a class is properly constructed could lead to many subtle bugs. If possible the class should be constructed to a known state so that it can be destructed without problems.

You should also save information about why the construction failed, which could be useful for future class operations. The future C++ exception handling scheme will solve this problem. Also one might use the MacApp `FailInfo` exception handling files in other non-MacApp projects.

Operator Overload Issues

Assignment operators should always start with a test that checks whether the object (by mistake) wants to assign to itself, as in `aFoo = aFoo`;, which could cause subtle problems. This is done as in the following:

```
TFoo& TFoo::operator=(const TFoo& aFoo)
{
       if (this == &aFoo) return *this;
         //...normal assignment duties...
         return *this;
}
```

Also, always overload all cases of the operator use, for instance both the '`x = x + y`' and the '`x += y`' operations.

If you are overloading certain operators, make sure that you know whether they have already been overloaded, and what they return/pass as values. Otherwise the compiler will complain about mismatch between formal and actual parameter types. For instance, `new` is overloaded with `PascalObject` and `HandleObject` base classes, and returns a `Handle` instead of a `void*`. Note also that if two programmers independently change the behavior of new, the resulting program might not work as expected.

Also, you need to inherit publicly from your base classes if you want the behavior of any new operator overload in the base class.

5. Inlining Issues

General

The C++ inlining feature is purely a hint to the compiler indicating that inline substitution of the function body is to be preferred to the usual function implementation. Inline code is usually used for code optimization: instead of calling a function, the whole body is inlined at the point of call, thus saving the cost of a function call.

Here's a simple example:

```
class TClass {
public:
       long GetField(void) {return this->fField;};
       void SetField(long);
private:
       long   fField;
};

inline void TClass::SetField(long theValue)
{
       this->fField = theValue;
}
```

Note that there are two different ways inline is indicated: by placing the function specifier inline in front of the function (or member function) declarator, or by defining the code directly in the class (by which the statements are automatically considered to be inlined). See Section 7.2.1 of *The Annotated C++ Reference Manual* for more information on inline function declarations.

As inlining is purely an optimization issue, it should be used only when the benefits in run-time or space outweigh the costs and inconveniences imposed by its use. The major cost of a function call is usually the cost of executing the function body, not the cost of making the call. Therefore, inlining should be used mostly for simple functions. Examples of such functions are functions that set or get a value, increment or decrement a value, or directly call another function. A function consisting of one or two simple expressions is usually a good inline candidate.

Compiler Considerations Concerning Inline Statements

The MPW C++ compiler has a set of rules by which it determines if an inline statement will be inlined or not. Some of the rules are easily quantified, such as the fact that recursive functions are never inlined; others vary depending upon whether or not the inline function return type is void, and upon the calling context. An inline function invoked in an expression context other than a call statement cannot be inlined if it contains code that cannot be reduced to one or more expressions. For instance, an `if-then-else` statement is only acceptable in such a calling context if it can be successfully converted to a conditional `(?:)` expression.

The following rules concern Apple's AT&T CFront port, MPW C++ 3.2 (and should also cover most cases with MPW C++ 3.1):

Recursive Functions

Recursive functions are never inlined.

Large Functions

Any function containing 12 or more assignments will not be inlined. Otherwise, size is less of an issue than complexity. For example, a function containing 5 or more calls will not be inlined, but the compiler may also refuse to inline a function containing fewer calls if there are other statements adding to the complexity. You can override the compiler's decision not to inline something based on size by using the -z17 option, but caution should be exercised.

Functions Invoked Before Defined

If an inline function is called before it is defined, it cannot be inlined. For example:

```
static int an_inline_function();

int an_outline()
{
        return (an_inline_function());
}

static inline int an_inline_function()
{
        return 1;
}
```

Because the compiler had not seen the inline body of "an_inline" when it encountered the first call, it will generate a call in "an_outline" and an out-of-line copy of "an_inline".

Functions Invoked Twice or More Within an Expression

Typically, in this case, the compiler will inline the body of the function for the first usage and then use calls for subsequent uses within the same expression. For instance:

```
        i = some_inline() + some_inline();
```

An out-of-line copy of "some_inline" will be generated and called for the right operand of the addition, in most cases. The compiler may still be able to inline the function in both places if it declares no variables and if either it has no parameters or the actual parameter expressions are sufficiently simple.

Functions Containing loop, switch, goto, label, break, or continue Statements

Value-returning inline functions will not be inlined if they contain any of the statement types listed above. Even non-value-returning inline functions cannot be inlined if they contain such statements and are invoked in the middle of an expression; the only control flow statement that can be inserted into the middle of an expression is the `if-then-else` statement.

Taking the Address of an Inline Function

An out-of-line copy will be generated for any inline function whose address is needed, either because it is the explicit target of the unary '&' operator or because it is used to initialize a function pointer. Virtual calls of virtual inline functions fall into this category as well.

Non-Value-Returning Inline Functions Containing a Return Statement

These are never inlined.

Functions Declaring Static Variables

These are never inlined.

Functions Containing Statements After a Return

An out-of-line copy will be generated for any inline function with one or more statements after the return statement. This applies primarily to value-returning functions, since non-value-returning functions containing any return statement will never be inlined. For example:

```
inline int an_inline(void)
{
        if (condition)
                return 0;
        do_something();                 // will suppress inlining
        return something;
}
```

Segmentation Issues Concerning Non-Inlined Statements—Which Segment Do Unexpectedly Outlined Functions Appear In?

Inlined code, which is suddenly outlined by the compiler, usually ends up in whichever segment that is actual for the call that caused the inline code to be outlined. Typically the outlined code ends up at the end of the object file.

If you want to control in what segment the code will be placed, bracket all the header files with `'#pragma segment HeaderFiles'`,in combination with `#pragma push` and `#pragma pop`. This way you are able to control into what segment the inline code will end in if it's suddenly outlined. Here's an example of how this is done:

```
// push the pragma state information
#pragma push
```

```
// define segment name for suddenly outlined inline-code
#pragma segment IfOutlinedItGoesHere

class TFoo{
public:
        TFoo(){/* ...*/}
        long  InlineMeMaybe(long x){/* ...*/}
// ...
};

// pop back the original pragma information
#pragma pop
```

Compiler Directives

- **Suppression of No Inline Code**

The MPW C++ compiler has a `-z0` switch, which forces all inline code to be non-inline. This switch is useful when trying to track down problems that are eventually related to inline code generation.

- **Forced Inlining of Large Functions**

The MPW C++ compiler has a `-z17` switch that will force inlining of functions that would normally be rejected because of size considerations. Consider carefully before using this switch as it can lead to large code. It may also cause CFront to generate expressions larger than the MPW C compiler can handle.

Warnings

The new MPW 3.2 C++ compiler (available on ETO CD #5 forward) is based on CFront 2.1 (AT&T), and the `-w` flag in this release will now indicate when the compiler chooses not to inline a function declared inline.

Conclusion

Inline-defined functions are just hints to the compiler, and the inline code generation rules will vary from implementation to implementation. The rules described in this document are true for the Apple MPW C++ compiler. Some of them are limitations resulting from the fact that MPW C++ generates C code; other inlining problems will also apply to native compilers. One needs to realize that inline statements are not always inlined by C++ compilers, and that inlining rules are C++ /C compiler implementation dependent.

6. Memory Leakage

General

Memory leakage usually occurs when space is dynamically allocated on the heap and, usually because of a programming error, the heap space is never deallocated. Unfortunately, with C++ hidden memory leaks can happen, which in the Macintosh memory system will trigger a heap-stack collision and a bomb. Here's a list of possible memory leaks and memory allocation problems, and ways to avoid them:

• Nonpaired New/Deletes

If you allocate data on the heap with new, it usually should be deleted with a subsequent delete call. This usually happens when the object goes out of scope, but if the data is explicitly allocated in the heap the compiler doesn't know how to purge this when the object goes out of scope. This problem comes up especially when an object creates space for data on the heap as part of its class structure, as in the following:

```
class TFoo{
public:
        TFoo(char* name);           // forgot to declare a ~TFoo() which would
                                    // delete the fName structure
private:
        char*   fName;
};

TFoo::TFoo(char* name)
{
        fName = new char[strlen(name) + 1];
        strcpy(fName, name);
}
```

The `fName` data structure will be on the heap until delete is called. If you delete the `fName` string in the destructor then you will avoid the memory leak.

• Object Pointers That Are Nested Inside Classes

If the class makes use of objects that are referenced via pointers, they need to be deleted; otherwise the data will stay in the heap, as in the following:

```
class TBar{
public:
        TBar(char* type);
        ~TBar();
private:
        TFoo*   fFoo;               // from the earlier example
        char*   fType;
};


TBar::TBar(char* type)
{
        fFoo = new TFoo("Willie");
        fType = new char[strlen(type) + 1];
        strcpy(fType, type);
}

TBar::~TBar()
{
        delete fType;          // this is OK
                                    // but you also need to delete the fFoo, as in:
   // delete fFoo;
}
```

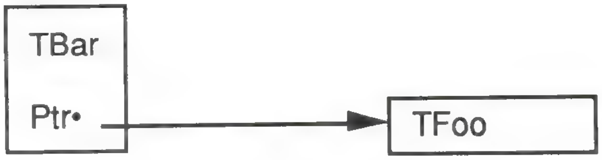


Figure 1 Nested Objects

• Missing Size Arguments to the Delete Function

The delete function needs the size of the deleted data structures, especially in the case of deletion or arrays of objects. Note that this problem will go away with MPW C++ compilers (MPW 3.2 C++ and later ones) where the general [] notation keeps track of the sizes of the arrays. For example:

```
main()
{
        TFoo* fooArray = new TFoo[10];
        // create an array of 10 TFoo:s
        // do something
        // delete the array
        delete [] fooArray;          // should be delete [10] fooArray with MPW C++ 3.1;
}
```

• Problems With Arrays of Pointers Versus Arrays of Objects

There is a subtle but important difference between an array of pointers to objects, and an array of objects themselves. The use of the delete operator is different in either case, as in the following:

```
main()
{
        TFoo** fooArray = new TFoo[10];         // array of pointers to objects

        for(int i=0;i<10;i++)                   // create the objects in the array
                fooArray[i] = new TFoo("Steve");

        // do something

        // now clean up the array
        delete [10] fooArray;                   // this only cleans up the
                                                // pointers, not the objects
                                                // themselves
        // the following code should be used instead:
        for(i=0;i<10;i++)
                delete fooArray[i];

        delete [] fooArray;

        return 0;
}
```

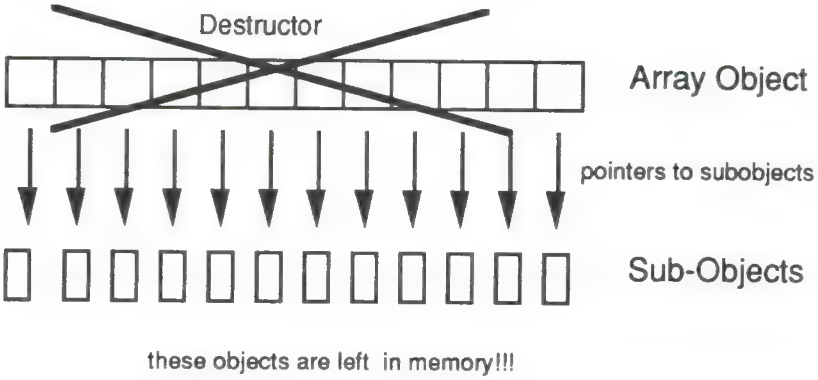


Figure 2 Objects Left Due to Missing Arguments

Memory leaks such as this becomes even more dangerous with object-oriented databases and persistence cases, where a leak could address more and more hard disk space on a server.

• Missing Copy Constructor

When operator overloading occurs, dynamically allocated memory for temporary data storage can suddenly develop a subtle leak that eats memory slowly. For instance an implicit call to an undefined copy constructor could be dangerous. These kinds of constructors are called whenever an initialization is done in code, when objects are passed by value on the stack, or when objects are returned by value. For example:

```
class TFoo{
public:
        TFoo(char* name, int age);
        // TFoo(const TFoo&);              note, no copy constructor defined!!
        ~TFoo();

        TFoo Copy(TFoo);                   // copy function, will call default
                                           // copy constructor
private:
        char*   fName;
        int     fAge;
};


TFoo::TFoo(char* name, int age)
{
        fName = new char[strlen(name) + 1];
        strcpy(fName,name);
        fAge = age;
}

TFoo::~TFoo()
{
        delete fName;
}

TFoo TFoo::Copy(TFoo orig)                 // note that this code is the same as the code
                                           // which the compiler would create for a default
                                           // copy constructor (i.e. field-wise copy).
{
        fAge  = orig.fAge;                 // plain pointer copy
        fName = orig.fName;
        return *this;
}
```

```
main()
{
        // create two objects
        TFoo f1("James", 25);
        TFoo f2("Michael", 29);

        TFoo f3 = f2;                   // this calls the copy constructor
                                        // TFoo f3(f2) would also trigger this
        // do something
        f1.Copy(f2);                    // this causes two implicit calls
                                        // to the default copy constructor

        // we have a problem, fName is deleted twice, once when f1 is destructed,
        // and the second time when d2 is destructed

        return 0;
}

// solution, create a specific copy constructor, as in:

TFoo::TFoo(const TFoo& orig)
{
        fAge = orig.fAge;
        fName = new char[strlen(orig.fName) + 1];
        strcpy(fName, orig.fName);
}
```

In general, if the class constructor assigns dynamic data, there should be a copy constructor that does the same as well. Note also that call by reference does not generate a copy constructor, so use of references is both faster and should generate fewer unexpected memory leak problems.

• Missing Overload Assignment Operator (`operator=`)

Every class that dynamically allocate storage for members should also have a defined overload assignment operator. If this operator is not clearly designed, there can be memory leaks due to assignment of dynamic data. For example,

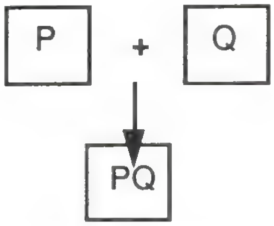
```
class TFoo{
public:
        TFoo(char* name, int age);
        TFoo(const TFoo&);
        // const TFoo& operator=(const TFoo& orig);        // note missing operator
                                                                   // overload operator
        ~TFoo();

private:
        char*   fName;
        int     fAge;
};


TFoo::TFoo(char* name, int age)
{
        fName = new char[strlen(name) + 1];
        strcpy(fName,name);
        fAge = age;
}
```

```
TFoo::~TFoo()
{
       delete fName;
}


TFoo::TFoo(const TFoo& orig)
{
       fAge = orig.fAge;
       fName = new char[strlen(orig.fName) + 1];
       strcpy(fName, orig.fName);
}


main()
{
       // create two objects
       TFoo f1("James", 25);
       TFoo f2("Michael", 29);

       // do something

       f2 = f1;                                  // this calls the default operator
                                                 // = overload, does not take into
                                                 // account the dynamic data (fName)

       return 0;
}

// The solution is to define an operator=:

const TFoo&
TFoo::operator=(const TFoo& orig)
{
       // avoid assignment to itself, as in aFoo = aFoo
       if(this ==&orig)                          // same address?
              return *this;

       fAge = orig.fAge;
       delete fName;                       // purge the dynamic memory slot
       fName = new char[strlen(orig.fName) + 1];
       strcpy(fName,orig.fName);

       return *this;
}
```

• Incorrectly Overloaded Operators

In general, try to make overloaded operators return references to objects to avoid overhead associated with calls to copy constructors. So how should you overload the operators in order to achieve this?

ex: P + Q; // '+' is overloaded



Don't change the internals of P and Q!

Figure 4 Correct Overload of Operators

Here's a good solution, we will return a real object instead of a reference in operator+:

```
class TFoo{
public:
        TFoo() {}
        TFoo(char* name, int age);
        TFoo(const TFoo&);
        const TFoo& operator=(const TFoo& orig);
        ~TFoo();
        // here's the example of operator+ overload:
        TFoo operator+(const TFoo&);                    // return TFoo by value!
                                                        // don't forget to overload += also!

private:
        char*   fName;
        int     fAge;
};


TFoo::TFoo(char* name, int age)
{
        fName = new char[strlen(name) + 1];
        strcpy(fName,name);
        fAge = age;
}

TFoo::~TFoo()
{
        delete fName;
}


TFoo::TFoo(const TFoo& orig)
{
        fAge = orig.fAge;
        fName = new char[strlen(orig.fName) + 1];
        strcpy(fName, orig.fName);
}

const TFoo&
TFoo::operator=(const TFoo& orig)
{
        // avoid assignment to itself, as in aFoo = aFoo
        if(this ==&orig)                                // same address?
                return *this;

        fAge = orig.fAge;
        delete fName;                       // purge the dynamic memory slot
        fName = new char[strlen(orig.fName) + 1];
        strcpy(fName,orig.fName);

        return *this;
}

TFoo
TFoo::operator+(const TFoo& orig)
{
        TFoo temp;                                 // create TFoo on the stack
        temp.fAge = fAge + orig.fAge;              // add ages, heh!

        temp.fName = new char[strlen(fName) + strlen(orig.fName) + 1];
        sprintf(temp.fName,"%s%s", fName, orig.fName);
                                                   // concatenate names, heh!
        return temp;
}
main()
{
        // create two objects
        TFoo f1("James", 25);
        TFoo f2("Michael", 29);

        TFoo f3 = f1 + f2;

        return 0;
}
```

Tricks to Help You Find Memory Leaks

In general, you need to go through the code carefully and analyze any possible subtle memory leaks. Another trick is to override the new and delete operators, and have them print status information to a log file (using for instance the __FILE__ and __LINE__ macros), and after running the program you can check to see whether each created data structure on the heap is deleted or not.

Here's an example of a possible tracer class, which could be used as the "stamp" for keeping track of class construction and destruction:

```
#include <stream.h>

#define TRACEPOINT __FILE__,__LINE__
// make use of the ANSI __FILE__ and __LINE__ macros

class TTracer {
public:
                        TTracer(const char* className, const char* = 0, int = 0);
        virtual         ~TTracer();
private:
        const char*     fLabel;
        const char*     fFile;
        int             fLine;
        static int      fReferenceCount;    // keep track of how many TTracers we
                                            // construct
};

TTracer::TTracer(const char* label, const char* file, int line) :
                        fLabel(label), fFile(file), fLine(line)
{
        fReferenceCount++;
        cerr    << "File " << fFile <<" ; Line " << fLine
```

```
                <<"  #+++ constructor event in " << fLabel
                << " (reference count = " << fReferenceCount << ")\n";
}


TTracer::~TTracer()
{
        fReferenceCount--;
        cerr    << "File " << fFile <<" ; Line " << fLine
                << "  #--- destructor  event in " << fLabel
                << " (reference count = " << fReferenceCount << ")\n";
}


int TTracer::fReferenceCount = 0;           // initialize with 0 value

TTracer gGlobalTracer("gGlobalTracer", TRACEPOINT);
// this will construct a global/universal tracer


// example of use:

void InvertPermutation(int* perm, int* inv, int max)
{
        TTracer autoTracer("InvertPermutation function", TRACEPOINT);

        if(perm && (new TTracer("temp", TRACEPOINT))
                // show TTracer in action
                && inv
                && (new TTracer("temp2", TRACEPOINT))
                // these two are never destructed = memory leak!
                && (max > 0))
        {
                        TTracer otherTracer("otherTracer", TRACEPOINT);

                        for(int i = 0; i < max; i++)
                        {
                                TTracer thirdTracer("iterationTracing...",TRACEPOINT);
                                inv[perm[i]] = i;
                        }
        }
}


// array declarations
int perm [] = {1, 2, 3, 6, 7};
int max = 5;


main()
{
        int* inv = new int[max];
        InvertPermutation(&perm[0], inv, max);

        return 0;
}
```

The result should look like this (note the output; you can double-click from MPW to get to the source code line in action):

```
File TTracer.cp ; Line 52  #+++ constructor event in InvertPermutation
                        function (reference count = 1)
File TTracer.cp ; Line 54  #+++ constructor event in temp (reference count =
                        2)
```

```
File TTracer.cp ; Line 56  #+++ constructor event in temp2 (reference count =
                      3)
File TTracer.cp ; Line 59  #+++ constructor event in otherTracer (reference
                      count = 4)
File TTracer.cp ; Line 62  #+++ constructor event in iterationTracing...
                      (reference count = 5)
File TTracer.cp ; Line 62  #--- destructor  event in iterationTracing...
                      (reference count = 4)
File TTracer.cp ; Line 62  #+++ constructor event in iterationTracing...
                      (reference count = 5)
File TTracer.cp ; Line 62  #--- destructor  event in iterationTracing...
                      (reference count = 4)
File TTracer.cp ; Line 62  #+++ constructor event in iterationTracing...
                      (reference count = 5)
File TTracer.cp ; Line 62  #--- destructor  event in iterationTracing...
                      (reference count = 4)
File TTracer.cp ; Line 62  #+++ constructor event in iterationTracing...
                      (reference count = 5)
File TTracer.cp ; Line 62  #--- destructor  event in iterationTracing...
                      (reference count = 4)
File TTracer.cp ; Line 62  #+++ constructor event in iterationTracing...
                      (reference count = 5)
File TTracer.cp ; Line 62  #--- destructor  event in iterationTracing...
                      (reference count = 4)
File TTracer.cp ; Line 59  #--- destructor  event in otherTracer (reference
                      count = 3)
File TTracer.cp ; Line 52  #--- destructor  event in InvertPermutation
                      function (reference count = 2)
```

7. Virtual Functions

Virtual Base Classes

As part of the multiple inheritance semantics, MPW C++ contains a feature called *virtual base classes*. As you can see in figure 5, if both class B and C are subclasses of A, and class D has both B and C as base classes, then D unfortunately will have two A's subobjects if A is not a virtual base class.

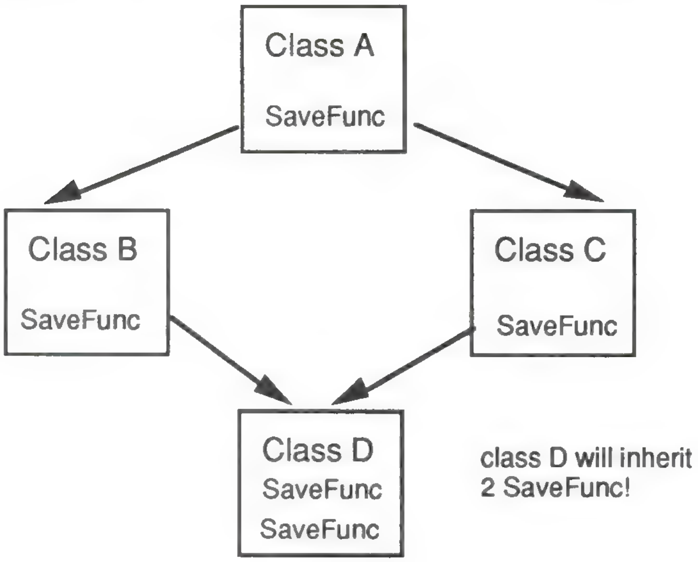


Figure 5 Virtual Base Classes

Try to avoid this confusing situation, because outside programmers might have a hard time trying to understand the new derived class. Also, virtual base classes have a problem: once you have a pointer to a virtual base, there is no way to convert it back into a pointer to its enclosing class.

So, if you have `TFoo` as a virtual base, and stick this class into an array or another collection, there's no way to convert it back to the right type via a cast when you get it out from the generic collection container.[1] Anyway, you should avoid casting base classes to derived classes if possible.

Also, see *Annotated C++ Reference Manual*, Section 10, for more information about virtual base classes.

Missing Virtual Functions

If you declare a virtual function in a class, you also need to implement the function. Otherwise the linker will complain about undefined entry, `name: (Error 28) "_ptbl_4TFoo"`, for example. This might happen if you define a function as virtual, but don't create the function until it's part of a subclass.

The exception to this is pure virtual functions.

Virtual Destructor Use

Destructors are not implicitly virtual whether the class has other virtual functions or not. This means that if you delete such an object via a pointer to one of its bases, the derived class destructors will not be called. This is bad, because it is important to call the right destructor.

[1] This problem will disappear with future template support.

If you wish the right destructor to be called during run-time, declare the destructor virtual. A good rule is to declare all destructors virtual by default, and deviate from this rule only if you don't want to have a vtable (that is, no other virtual functions in the class), or if you want to save some run-time lookup by providing a simple class.

Virtual Functions Are Not Real Functions

Virtual functions are references to virtual function resolve to vtable entries. Be aware that they are not similar to normal functions in all cases; for instance, you can't use them when unloading segments, as in

```
UnloadSeg((ProcPtr)&(TFoo::Method));
```

The workaround is to place an empty function stub in the same segment, and use this function name when calling `UnloadSeg`.

8. Compiler Issues

Declarations

The definition of C++ requires that data structures and functions have to be declared before they are used.Understanding this should eliminate a lot of obscure syntax problems. Note that when writing a particular class at the beginning of the header file you can use the class before the class is defined, as in the following:

```
class TFoo;                    // forward declare this class

class TBar{
// ...
        TFoo* fFooPtr;      // use the class!
// ...
};
```

Also, if you are using an enum or typedef in the class, it has to be defined before used, as in the following:

```
class TFoo{
public:
//      Constructors/Destructors
        TFoo();
        const TFoo& TFoo(const TFoo&);
        virtual ~TFoo();

//      Enums and Typedefs
        enum EPriority {kLow, kMedium, kHigh};

//      Accessors and mutators
        TFoo& SetPriority(EPriority);
        EPriority GetPriority();
// ...
};
```

Exception Handling and Register Optimization

The MPW C++/C compiler usually tries to move frequently updated variables to registers. This is important to know if you are using exception handling, either the MacApp provided calls or something based on setting/restoring registers after an exception has occurred.

The following piece of code shows the problem:

```
void ProblemCase(void)
{
        int     nCount;
        int     nElements;
        TFoo*   temp;

        TRY
        {
                for(nCount= 0; nCount < nElements; ++nCount){
                        temp = new TFoo;
                        temp->Initialize();
                        gApplication->AddTFoo(temp);
                }
        }
        RECOVER
        {
                if(temp != NULL) temp->Free();      // clean up
                if(count == 0) ExitApplication()    // exit application
        }
        ENDTRY
}
```

In this case the `nCount` integer and the `temp` pointer will most likely be optimized into a register allocation. If an exception occurs while the count it updated inside the register, there's no way for the exception handler to roll back the old values, because it assumes the stack based values are OK. Thus any `RECOVER` action that assumes that the values are OK might not work as expected.

Unfortunately MPW 3.2 C++ has not implemented the `volatile` keyword (because it requires a full implementation). However we can emulate the volatile behavior with a macro. We are interested in making sure the changed variable is never placed into a register:

```
#define VOLATILE(a) ((void) &a)
```

What we need to do is to make sure any possible variable that is subject to change is wrapped inside the `VOLATILE` macro before it's used inside `TRY/RECOVER` , as in

```
// ...

        VOLATILE(nCount);
        VOLATILE(temp);

        TRY
        {
                for(nCount= 0; nCount < nElements; ++nCount){
                        temp = new TFoo;
// .....
```

9. Testing/Debugging

General Issues

Do empirical testing/debugging sessions; eliminate one module at a time until you have pinpointed the problem. Write incremental code, and test the new features before continuing with the code writing.

Don't change too many variables at once when you are testing the code. All in all, a controlled test experiment helps you understand how certain parts interact with each other. If possible, use debugging code that can be turned on and off with a compiler flag.

10. Conclusion

Careful consideration of any possible side effects will help a lot when using any computer language. A good motto for programmers is *Prepare for the worst, and plan for the best.*

Further Reference:

- MPW C++ 3.1 Reference
- MPW C++ 3.1 Release Notes
- *The Annotated C++ Reference Manual,* Ellis and Stroustrup, Addison-Wesley
- *C Traps and Pitfalls,* A. Koenig, Addison-Wesley

#310: Who Put That Resource in My CDEV?

Written by: C. K. Haun <TR>

February 1992

This Technical Note discusses the new 'fwst' resource added to some Control Panels under System 7.0 and later.

A New World for Control Panels

System 7 changes many of the rules for Control Panels (cdevs), and these changes are very well documented in Chapter 10 of *Inside Macintosh* Volume VI. However, there is one thing *not* documented in *IM* VI that you need to be aware of, as it could cause you confusion and frustration— the 'fwst' resource.

Two groups of developers should be interested in this information: Control Panel developers and developers of virus and disk security software.

What Is This 'fwst' Thing in My CDev?!?!?!

OK, we admit it. The System can add a resource to your Control Panel. But it does it for a good reason, really!

Control Panels do not show up in the Control Panel desk accessory anymore. Each Control Panel a user opens will show up in its own window. Because of this, the Finder needs to have a way to remember (among other things) the position of the Control Panel window on the user's desktop so that the Finder can position the Control Panel in the same location every time the user opens it, thereby saving the user from having to continually reposition the window.

A new resource type—the 'fwst' resource—was created to keep track of the Control Panel window position (and other things). The contents of this resource is private; you should make no assumptions about the contents, size, or use of the components of the 'fwst' resource. The only public aspect of this resource is that it is used by the Finder to position a Control Panel window on the desktop.

The 'fwst' resource does not automatically get added to your Control Panel. If a user opens your Control Panel and closes it without moving the Control Panel window, then no 'fwst' resource is needed, since the default position for the window has not changed.

However, if the user moves the window and closes the Control Panel, a 'fwst' resource is added. This tells the Finder where to place the Control Panel window when the user opens it up again. This obviously is a very user-friendly thing to do. Users get consistent positioning of their windows, and are not frustrated by having to shuffle windows all the time. Note, however, that it could cause problems for you if you don't know that it may show up.

If you check your own resource fork for any reason (for example, scanning for viruses) you need to know that the 'fwst' resource may be there. If it's there, that is normal, and you should not treat that as a damages resource fork or a viral infection. If you notice the 'fwst' resource being added to a Control Panel and if you are a virus protection or disk security software developer you should not alert the user that a resource has been added or that a viral attack is taking place.

One More Thing

The presence of an 'fwst' resource has one more effect that you may find *very* frustrating, since until you know about it you can't figure it out. Another resource that you normally add to a Control Panel is the 'nrct' resource. This resource is used to specify a list of rectangles that your Control Panel used in the pre–System 7 Control Panel desk accessory.

The 'nrct' rectangle resource is described in *Inside Macintosh* Volume V, and the removal of the size restriction is documented in *IM* VI. Basically, what *IM* VI says is that now, since each Control Panel has its own window, your 'nrct' does not have to fit inside the old Control Panel bounding rectangle. Your Control Panel under System 7 can be much bigger than it was in any previous system. One thing that *IM* VI doesn't explicitly say is that the first rectangle in your 'nrct' resource is the bounding rectangle for the Control Panel window under System 7 and later.

What Does This Have to Do With the 'fwst'?

The 'fwst' **takes precedence** over the 'nrct'. So, if you have a 'fwst' in your Control Panel, any changes to the first rectangle in your 'nrct' will *not* be recognized!

This can be very frustrating during Control Panel development. You've been merrily debugging your Control Panel, moving it around the Finder, and making sure everything works. You decide you need to add another item to the Control Panel, and therefore you want the Control Panel window to be bigger. "Great," think you,"System 7 doesn't care how big I make it!" and you go into ResEdit and change your 'nrct'. You go back to the Finder, open the Control Panel, and nothing has changed!

What's happening is that the 'fwst' is overriding the 'nrct'. If you need to change the 'nrct' of your Control Panel, make *sure* you check to see if there is an 'fwst' resource in your Control Panel's resource fork. If there is, delete it and make the necessary changes to your 'nrct'. With no 'fwst', your 'nrct' values will be recognized, and a new 'fwst' reflecting the correct rectangle will be created if the window is moved. Use ResEdit, your favorite resource editor, or Rez in MPW to remove the 'fwst'. Here's a command line you can add to MPW build script for your Control Panel that will remove the 'fwst' resource from the Control Panel automatically during the build process;

```
echo "delete 'fwst';" | Rez -a -o "{MyControlPanelName}"
```

Of course, replace `"{MyControlPanelName}"` with whatever name or variable you are using in your build script to identify your Control Panel.

And remember to remove whatever 'fwst' is in the Control Panel before your ship your product. This will let the Control Panel come up on your user's machines in the default location, and the user can decide where he or she would like the Control Panel placed.

Further Reference:

- *Inside Macintosh*, Volume V, The Control Panel
- *Inside Macintosh*, Volume VI, Control Panels

#311: What's New With AppleTalk Phase 2

Updated by: Rich Kubota April 1992
Written by: Rich Kubota & Scott Kuechle February 1992

This Technical Note discusses the new features of AppleTalk available for System 7.0 and AppleTalk version 57. The new features include support for the Flagship Naming Service and the AppleTalk Multiple Node Architecture. We present the Multiple Node Architecture and discuss the new calls available to applications. We also discuss the impact of the new architecture on AppleTalk Device files (ADEVs), and the changes necessary to make them multinode compatible. Finally, we discuss the Flagship Naming Service, along with the new AppleTalk Transitions. The new transitions notify a process of changes to the Flagship name, network cable range, router status, and processor speed.

Changes since February 1992: Provided additional detail on the implementation to the AAddNode, ADelNode, and AGetNodeRef calls including parameter offsets. Added sample code to check for existence of LAP Manager. Added Pascal source to determine whether the LAP Manager exists. Added warning to check the result from the LAPAddATQ function since the System 7 Tuner extension may not load AppleTalk resources. Corrected typographical errors. Added information on the discussion on the Speed Change AppleTalk Transition event. Added discussion regarding the 'atkv' gestalt selector. Sidebars highlight changes or additions to this document.

AppleTalk Multiple Node Architecture

Supporting multiple node addresses on a single machine connected to AppleTalk is a feature that has been created to support software applications such as AppleTalk Remote Access. Its implementation is general enough to be used by other applications as well.

Note: AppleTalk version 57 is required to support the AppleTalk Multiple Node Architecture. Version 57 is compatible with System 6.0.4 and greater. If you implement multinode functionality into your program you should also plan to include AppleTalk version 57 with your product. Contact Apple's Software Licensing department for information on licensing version 57. For testing purposes, AppleTalk version 57 can be installed by using the Network Software Installer v1.1, now available on the latest Developer CD, on AppleLink in the Developer Services Bulletin Board, and on the Internet through anonymous FTP to ftp.apple.com (130.43.2.3).

Software Licensing can be reached as follows:

Software Licensing
Apple Computer, Inc.
20525 Mariani Avenue, M/S 38-I
Cupertino, CA 95014
MCI: 312-5360
AppleLink: SW.LICENSE
Internet: SW.LICENSE@AppleLink.Apple.com
(408) 974-4667

What Is It?

Multiple Node AppleTalk provides network node addresses that are in addition to the normal (user node) DDP address assigned when AppleTalk is opened. These additional addresses have different characteristics from those of the user node address. They are not connected to the protocol stack above the data link layer. When an application acquires a multinode, the application has to supply a receive routine through which AppleTalk will deliver broadcasts and packets directed to that multinode address.

The number of multinode addresses that can be supported on one single machine is determined by a static limit imposed by the AppleTalk ADEV itself (for example, EtherTalk). The limit is currently 253 nodes for Apple's implementation of EtherTalk ($0, $FF, and $FE being invalid node addresses) and 254 for LocalTalk ($0 and $FF being invalid node addresses). The number of receive routines that .MPP supports is determined by the static limit of 256. If all of the multiple nodes acquired need to have unique receive routines, then only a maximum of 256 nodes can be acquired, even if the ADEV provides support for more than 256 nodes. .MPP will support the lesser of either the maximum of 256 receive routines, or the node limit imposed by the ADEV.

Outbound DDP packets can be created with a user-specified **source network, node,** and **socket** (normally equal to a multinode address) with the new Network Write call. With this capability and the packet reception rules described above, a single machine can effectively become several nodes on a network. The **user** node continues to function as it always has.

Things You Need to Know When Writing a Multinode Application

Two new .MPP driver control calls have been added to allow multinode applications to add and remove multinodes.

AddNode (csCode=262)

A user can request an extra node using a control call to the .MPP driver after it has opened. Only one node is acquired through each call.

```
Parameter Block:
        -->     24      ioRefNum        short           ; driver ref. number
        -->     26      csCode          short           ; always = AddNode (262)
        -->     36      reqNodeAddr     AddrBlock       ; the preferred address requested
                                                        ; by the user.
        <--     40      actNodeAddr     AddrBlock       ; actual node address acquired.
        -->     44      recvRoutine     long            ; address of the receive routine for MPP
                                                        ; to call during packet delivery
        -->     48      reqCableLo      short           ; the preferred range for the
        -->     50      reqCableHi      short           ; node being acquired.
        -->     52      reserved[70]    char            ; 70 reserved bytes

AddrBlock:
        aNet            short                           ; network #
        aNode           unsigned char                   ; node #
        aSocket         unsigned char                   ; should be zero for this call.
```

This `.MPP AddNode` call must be made as an IMMEDIATE control call at system task time. The result code will be returned in the `ioResult` field in the parameter block. The result code –1021 indicates that the .MPP driver was unable to continue with the `AddNode` call because of the current state of .MPP. The caller should retry the AddNode call (the retry can be issued immediately after the AddNode call failed with –1021) until a node address is successfully attained or another error is returned. The .MPP driver will try to acquire the requested node address.

If the requested node address is zero, invalid, or already taken by another machine on the network, a random node address will be generated by the .MPP driver. The parameters reqCableLo and reqCableHi will be used only if there is no router on the network and all the node addresses in the network number specified in NetHint (the last used network number stored in parameter RAM) are taken up.

In this case, the .MPP driver will try to acquire a node address from the network range as specified by reqCableLo and reqCableHi. The network range is defined by the seed router on a network. If a specific cable range is not important to the application, set the reqCableLo and reqCableHi fields to zero. The recvRoutine is an address of a routine in the application to receive broadcasts and directed packets for the corresponding multinode.

Possible Error Codes:

```
        noErr                  0             ; success
        tryAddNodeAgainErr     -1021         ; .MPP was not able to add node, try again.
        MNNotSupported         -1022         ; Multinode is not supported by
                                             ; the current ADEV
        noMoreMultiNodes       -1023         ; no node address is available on
                                             ; the network
```

RemoveNode (csCode=263)

This call removes a multinode address and must be made at system task time. Removal of the user node is not allowed.

```
Parameter Block:
        -->     24      ioRefNum        word            ; driver ref. number
        -->     26      csCode          word            ; always = RemoveNode (263)
        -->     36      NodeAddr        AddrBlock       ; node address to be deleted.
```

Possible Error Codes:

```
        noErr           0               ; success
        paramErr        -50             ; bad parameter passed
```

Receiving Packets

Broadcast packets are delivered to both the user's node and the multinodes on every machine. If several multinodes are acquired with the same recvRoutine address, the recvRoutine, listening for these multinodes, will only be called once in the case of a broadcast packet.

Multinode receive handlers should determine the number of bytes already read into the Read Header Area (RHA) by subtracting the beginning address of the RHA from the value in A3 (see *Inside Macintosh* Volume II, page 326, for a description of the Read Header Area). A3 points past the last byte read in the RHA. The offset of RHA from the top of the .MPP variables is defined by the equate ToRHA in the MPW include file ATalkEqu.a. The receive handler is expected to call ReadRest to read in the rest of the packet. In the case of LocalTalk, ReadRest should be done as soon as possible to avoid loss of the packet. Register A4 contains the pointer to the ReadPacket and ReadRest routines in the ADEV.

To read in the rest of the packet:

```
          JSR   2(A4)
```

On entry:

A3	pointer to a buffer to hold the bytes
D3	size of the buffer (word), which can be zero to throw away packet

On exit:

D0	modified
D1	modified
D2	preserved
D3	Equals zero if requested number of bytes were read; is less than zero if packet was –D3 bytes too large to fit in buffer and was truncated; is greater than zero if D3 bytes were not read (packet is smaller than buffer)
A0	preserved
A1	preserved
A2	preserved
A3	pointer to 1 byte after the last byte read

For more information about `ReadPacket` and `ReadRest`, refer to the *Inside Macintosh* Volume II, page 327.

A user can determine if a link is extended by using the `GetAppleTalkInfo` control call. The `Configuration` field returned by this call is a 32-bit word that describes the AppleTalk configuration. Bit number 15 (0 is LSB) is on if the link in use is extended. Refer to *Inside Macintosh* Volume VI, page 32-15.

Sending Datagrams Through Multinodes

To send packets through multinodes, use the new .MPP control call, `NetWrite`. `NetWrite` allows the owner of the multinode to specify a **source network, node,** and **socket** from which to send a datagram.

NetWrite (csCode=261)

```
Parameter Block:
       -->     26     csCode         word          ; always NetWrite (261)
       -->     29     checkSumFlag   byte          ; checksum flag
       -->     30     wdsPointer     pointer       ; write data structure
```

Possible Error Codes:

```
       noErr          0             ; success
       ddpLenErr      -92           ; datagram length too big
       noBridgeErr    -93           ; no router found
       excessCollsns  -95           ; excessive collisions on write
```

This call is very similar to the `WriteDDP` call. The key differences are as follows:

- The source socket is not specified in the parameter block. Instead it is specified along with the source network number and source node address in the DDP header pointed to by the write data structure. Furthermore, the socket need not be opened. Refer to *Inside Macintosh* Volume II, page 310, for a description of the Write Data Structure, WDS. It is important to note that the caller needs to fill in the WDS with the source network, source node, and source socket values. .MPP does not set these values for the NetWrite call.

- The `checkSumFlag` field has a slightly different meaning. If true (nonzero), then the checksum for the datagram will be calculated prior to transmission and placed into the DDP header of the packet. If false (zero), then the **checksum field is left alone** in the DDP header portion of the packet. Thus if a checksum is already present, it is passed along unmodified. For example, the AppleTalk Remote Access program sets this field to zero, since remote packets that it passes to the .MPP driver already have valid checksum fields. Finally, if the application desires no checksum, the checksum field in the DDP header in the WDS header must be set to zero.

Datagrams sent with this call are *always sent using a long DDP header*. Refer to *Inside AppleTalk*, Second Edition, page 4-16, for a description of the DDP header. Even if the destination node is on the same LocalTalk network, a long DDP datagram is used so that the source information can be specified. The LAP header source node field will always be equal to the user node address (`sysLapAddr`), regardless of the source node address in the DDP header.

AppleTalk Remote Access Network Number Remapping

Network applications should be careful not to pass network numbers as data in a network transaction. AppleTalk Remote Access performs limited network number remapping. If network numbers are passed as data, they will not get remapped. AppleTalk Remote Access recognizes network numbers in the DDP header and among the various standard protocol packets, NBP, ZIP, RTMP, and so on.

Is There a Router on the Network?

Do not assume that there are no routers on the network if your network number is zero. With AppleTalk Remote Access, you can be on network zero and be connected to a remote network. Network applications should look at the `SysABridge` low-memory global or use the `GetZoneList` or the `GetBridgeAddress` calls to determine if there is a router on the network.

New for AppleTalk ADEVs

Several new calls have been implemented into the .MPP driver for AppleTalk version 57. Two calls, `AOpen` and `AClose`, were built into AppleTalk version 54 and greater, and are also documented here. These calls notified the ADEV of changes in the status of the .MPP driver. For AppleTalk version 57, three new calls, `AAddNode, ADelNode, and AGetNodeRef`, plus a change to the `AGetInfo` call, were implemented to support the Multiple Node Architecture.

EtherTalk Phase 2, version 2.3, and TokenTalk phase 2, version 2.4, drivers support the new Multiple Node Architecture. Both drivers and AppleTalk version 57, are available through the Network Software Installer, version 1.1. As mentioned previously, AppleTalk version 57 and these drivers, are compatible with System 6.0.4 and greater. Note that the AppleTalk Remote Access product includes the EtherTalk Phase 2, version 2.3 driver, but *not* the multinode-compatible TokenTalk Phase 2, version 2.4, driver. Token Ring developers, who license

TokenTalk Phase 2, version 2.2 and earlier, should contact Apple's Software Licensing department .

The following information describes changes to the ADEV that are required for multinode compatibility. This information is of specific importance to developers of custom ADEVs. The ADEV can be expected to function under System 6.0.4 and greater. A version 3 ADEV **must** be used with AppleTalk version 57 or greater. Developers of custom ADEVs will want to contact Software Licensing to license AppleTalk version 57.

For compatibility with Multinode AppleTalk, the `'atlk'` resource of an ADEV must be modified to respond to these calls as described below. To determine whether an ADEV is multinode compatible, the .MPP driver makes an `AGetInfo` call to determine whether the ADEV version is 3 or greater. Any ADEVs responding with a version of 3 or greater must be prepared to respond to the new calls: `AAddNode, ADelNode, and AGetNodeRef.` See the *Macintosh AppleTalk Connections Programmer's Guide* for more information about writing an AppleTalk ADEV.

The desired architecture for a multinode-compatible ADEV is such that it delivers incoming packets to the LAP Manager along with an address reference number, `AddrRefNum.` The LAP Manager uses the `AddrRefNum` to locate the correct receive routine to process the packet. For broadcast packets, the LAP Manager handles multiple deliveries of the packet to each multinode receive routine.

The .MPP driver for AppleTalk version 57 supports the new control call to add and remove multinodes, along with the network write call which allows the specification of the source address. .MPP includes a modification in its write function to check for one multinode sending to another. .MPP supports inter-multinode transmission within the same machine. For example, the user node may want to send a packet to a multinode within the same system.

AGetInfo (D0=3)

The `AGetInfo` call should be modified to return the maximum number of **AppleTalk** nodes that can be provided by the atlk. This limit will be used by .MPP to control the number of multinodes that can be added on a single machine. The new interface is as follows:

Call:	`D1` (word)	length (in bytes) of reply buffer
	`A1` ->	Ptr to GetInfo record buffer
Return:	`A1` ->	Ptr to GetInfo record
	`D0`	nonzero if error (buffer is too small)

```
       AGetInfoRec = RECORD
<--    version:              INTEGER;        { version of ADEV, set to three (3) }
<--    length:               INTEGER;        { length of this record in bytes }
<--    speed:                LongInt;        { speed of link in bits/sec }
<--    BandWidth:            Byte;           { link speed weight factor }
<--    reserved:             Byte;           { set to zero }
<--    reserved:             Byte;           { set to zero }
<--    reserved:             Byte;           { set to zero }
<--    flags:                Byte;           { see below }
<--    linkAddrSize:         Byte;           { of link addr in bytes }
<--    linkAddress:          ARRAY[0..5] OF Byte;
<--    maxnodes:             INTEGER;
       END;

       flags: bit 7 = 1 if this is an extended AppleTalk, else 0
              bit 6 = 1 if the link is used for a router-only connection (reserved
                    for half-routing)
              bit 5 through 0 reserved, = 0
```

maxnodes is the total number of nodes (user node and multinodes) the ADEV supports. If a version 3 ADEV does not support multinodes, it must return 0 or 1 in the maxnodes field in AGetInfoRec and the ADEV will not be called to acquire multinodes. The version 3 ADEV will be called by .MPP in one of the following two ways to acquire the user node:

- if the ADEV returns a value of 0 in maxnodes, .MPP will issue Lap Write calls to the ADEV with D0 set to $FF indicating that ENQs should be sent to acquire the user node. .MPP is responsible for retries of ENQs to make sure no other nodes on the network already have this address. This was the method .MPP used to acquire the user node before multinodes were introduced. This method of sending ENQs must be available, even though the new AAddNode call is provided, to allow older versions of AppleTalk to function properly with a version 3 ADEV.

- if the ADEV returns a value of 1 in maxnodes, the new AAddNode function will be called by .MPP to acquire the user node.

For values of maxnode greater than 1, the new AAddNode function will be called by .MPP to acquire the additional multinodes.

AAddNode (D0=9)

This is a **new** call which is used to request the acquisition of an AppleTalk node address. It is called by the .MPP driver during the execution of the AddNode control call mentioned earlier. The ADEV is responsible for retrying enough ENQs to make sure no other nodes on the network already have the address. .MPP will make this call only during system task time.

Call:	A0->	parameter block
Return:	D0	= zero if address was acquired successfully ≠ zero if no more addresses can be acquired

```
        atlkPBRec           Record  csParam
-->     NetAddr             DS.L    1       ; offset 0x1C  24-bit node address to acquire
-->     NumTrys             DS.W    1       ; offset 0x20  # of tries for address
-->     DRVRPtr             DS.L    1       ; offset 0x22  ptr to .MPP vars
-->     PortUsePtr          DS.L    1       ; offset 0x26  ptr to port use byte
-->     AddrRefNum          DS.W    1       ; offset 0x2A  address ref number used by .MPP
                            EndR
```

The offset values describe the location of the fields from the beginning of the parameter block pointed to by A0. atlkPBRec is the standard parameter block record header for a _Control call. The field NetAddr is the 24-bit AppleTalk node address that should be acquired. The node number is in the least significant byte 0 of NetAddr. The network number is in bytes 1 and 2 of NetAddr; byte 3 is unused. NumTrys is the number of tries the atlk should send AARP probes on non-LocalTalk networks to verify that the address is not in use by another entity. On LocalTalk networks, NumTrys x 32 number of ENQs will be sent to verify an address.

DRVRPtr and PortUsePtr are normally passed when the atlk is called to perform a write function. For ADEVs that support multinodes, AppleTalk calls the new AAddNode function rather than the write function in the ADEV to send ENQs to acquire nodes. However, the values DRVRPtr and PortUsePtr are still required for the ADEV to function properly and are passed to the AAddNode call. AddrRefNum is a reference number passed in by .MPP. The ADEV must store each reference number with its corresponding multinode address. The use of the reference number is described in the following two sections.

For multinode-compatible ADEVs, .MPP will issue the first AAddNode call to acquire the user node. The AddrRefNum associated with the user node must be 0xFFFF. It is important to assign 0xFFFF as the AddrRefNum of the user node, and to disregard the AddrRefNum passed by .MPP for the user node. See the discussion at the end of the ADelNode description.

ADelNode (D0=10)

This is a **new** call which is used to remove an AppleTalk node address. It may be called by the .MPP driver to process the RemoveNode control call mentioned earlier.

Call: A0-> parameter block
NetAddr contains the node address to be deleted
Return: D0 = zero if address is removed successfully
≠ zero if address does not exist
atlkPBRec.AddrRefNum = AddrRefNum to be used by .MPP if the operation is successful

```
        atlkPBRec               Record  csParam
-->     NetAddr                 DS.L    1       ; offset 0x1C  24-bit node address to remove
<--     AddrRefNum              DS.W    1       ; offset 0x2A  AddrRefNum passed in by AAddNode
                                                ;              on return
                                EndR
```

The field NetAddr is the 24-bit AppleTalk node address that should be removed. As with the AAddNode selector, the node number is in the least significant byte 0 of NetAddr. The network number is in bytes 1 and 2 of NetAddr; byte 3 is unused. The address reference number, AddrRefNum, associated with the NetAddr, must be returned to .MPP in order for .MPP to clean up its data structures for the removed node address.

As mentioned above, a value of 0xFFFF must be returned to .MPP after deleting the user node. When the AppleTalk connection is started up for the first time on an extended network, the ADEV can expect to process an AAddNode request followed shortly by an ADelNode request. This results from the implementation of the provisional node address for the purpose of talking with the router to determine the valid network number range to which the node is connected. After obtaining the network range, .MPP issues the ADelNode call to delete the provisional node. The next AAddNode call will be to acquire the unique node ID for the user node. As mentioned previously, .MPP may pass a value different than 0XFFFF for the user node. The user node is acquired before any multi-node. The ADEV needs to keep track of the number of AAddNode and ADelNode calls issued to determine whether the user node is being acquired. Refer to *Inside AppleTalk*, Second Edition, page 4-8, for additional information.

AGetNodeRef (D0=11)

This is a **new** call which is used by .MPP to find out if a multinode address exists on the current ADEV. This call is currently used by .MPP to check if a write should be looped back to one of the other nodes on the machine (the packet does not actually need to be sent through the network) or should be sent to the ADEV for transmission.

Call: A0-> parameter block
Return: D0-> = zero if address does not exist on this machine
≠ zero if address exists on this machine
atlkPBRec.AddrRefNum = AddrRefNum (corresponding to the node address) if the operation is successful

```
        atlkPBRec              Record csParam
-->     NetAddr                DS.L    1       ; offset 0x1C  24-bit node address to remove
<--     AddrRefNum             DS.W    1       ; offset 0x2A  AddrRefNum passed in by AAddNode
                                               ;               on return
                               EndR
```

The field `NetAddr` is the 24-bit AppleTalk node address whose `AddrRefNum` is requested. The node number is in the least significant byte 0 of `NetAddr`. The network number is in bytes 1 and 2 of `NetAddr`; byte 3 is unused. The address reference number, `AddrRefNum`, associated with the `NetAddr`, must be returned to .MPP. Remember to return 0xFFFF as the `AddrRefNum` for the user node.

AOpen (D0=7)

Call:

```
        -->     D4.B            current port number
```

ADEVs should expect the AOpen call whenever the .MPP driver is being opened. This is a good time for the ADEV to register multicast addresses with the link layer. After this call is completed, .MPP is ready to receive packets. If the ADEV does not process this message, simply return, `RTN`.

Note that AOpen is not specific to the Multinode Architecture.

AClose (D0=8)

AClose is called only when .MPP is being closed (for example, .MPP is closed when the "inactive" option is selected in the Chooser or when the user switches links in Network CDEV). The ADEV should deregister any multicast addresses with the link layer at this time. After this `AClose` call is completed, the ADEV should not defend for any node addresses until .MPP reopens and acquires new node addresses. If the ADEV does not process this message, simply return, `RTN`.

Note that `AClose` is not specific to the Multinode Architecture.

For comparison, descriptions of `AInstall` and `AShutDown` are documented as follows:

AInstall (D0=1)

Call:

```
        -->     D1.L = value from PRAM (slot, ID, unused, atlk resource ID)
        <--     D1.L = high 3 bytes for parameter RAM returned by the ADEV,
                       if no error
        <--     D0.W = error code
```

The `AInstall` call is made before .MPP is opened either during boot time or when the user switches links in Network CDEV. This call is made during system task time so that the ADEV is allowed to allocate memory, make file system calls, or load resources and so on. Note: `AOpen` call will be made during .MPP opens.

AShutDown (D0=2)

ADEVs should expect the `AShutDown` call to be made when the user switches links in the Network CDEV. The Network CDEV closes .MPP, which causes the `AClose` call to be made before the CDEV issues the `AShutDown` call. Note: the `AShutDown` call is always made during system task time; therefore, deleting memory, unloading resources, and file system calls can be done at this time.

Receiving Packets

The address reference number (`AddrRefNum`) associated with each node address must be passed to .MPP when delivering packets upward. When making the LAP Manager call `LReadDispatch` to deliver packets to AppleTalk, the ADEV must fill the high word of `D2` in with the address reference number, corresponding to the packet's destination address (LAP node address in the LocalTalk case and DDP address in the non-LocalTalk case). There are a few special cases:

- In the case of broadcasts and packets directed to the user node, $FFFF (word) should be used as the address reference number.
- On non-LocalTalk networks, packets with DDP destination addresses matching neither the user node address nor any of the multinode addresses should still be delivered to the LAP Manager so that the router can forward the packet on to the appropriate network. In this case, high word of `D2` should be filled in with the address reference number, $FFFE, to indicate to MPP that this packet is not for any of the nodes on the machine in the case of a router running on a machine on an extended network.
- On LocalTalk networks, the ADEV looks only at the LAP address; therefore, if the LAP address is not the user node, one of the multinodes, or a broadcast, the packet should be thrown away.

Defending Multinode Addresses

Both LocalTalk (RTS and CTS) and non-LocalTalk (AARP) ADEVs have to be modified to defend not only for the user node address but also for any active multinode addresses.

The 'atkv' Gestalt Selector

The 'atkv' Gestalt selector is available beginning with AppleTalk version 56 to provide more complete version information regarding AppleTalk, and as an alternative to the existing 'atlk' gestalt selector. Beginning with AppleTalk version 54, the 'atlk' Gestalt selector was available to provide basic version information. The 'atlk' selector is not available when AppleTalk is turned off in the Chooser. It is important to note that the information between the two resources are provided in a different manner. Calling Gestalt with the 'atlk' selector provides the major revision version information in the low order byte of the function result. Calling Gestalt with the 'atkv' selector provides the version information in a manner similar to the 'vers' resource. The format of the LONGINT result is as follows:

```
byte;                                    /* Major revision */
byte;                                    /* Minor revision */
byte         development = 0x20,         /* Release stage  */
             alpha = 0x40,
             beta = 0x60,
             final = 0x80, /* or */ release = 0x80;
byte;                                    /* Non-final release #  */
```

For example, passing the 'atkv' selector in a Gestalt call under AppleTalk 57, gives the following LONGINT result: 0x39000800.

With the release of the System 7 Tuner product, AppleTalk may not be loaded at startup, if prior to the previous shutdown, AppleTalk was turned off in the Chooser. Under this circumstance, the 'atkv' selector is not available. If the 'atkv' selector is not available under System 7, this is an indicator that AppleTalk cannot be turned without doing so in the Chooser and rebooting the system.

The AppleTalk Transition Queue

The AppleTalk transition queue keeps applications and other resident processes on the Macintosh informed of AppleTalk events, such as the opening and closing of AppleTalk drivers, or changes to the Flagship name (to be discussed later in this Note). A comprehensive discussion of the AppleTalk Transition Queue is presented in *Inside Macintosh* Volume VI, Chapter 32. New to the AppleTalk Transition Queue are messages regarding the Flagship Naming Service, the AppleTalk Multiple Node Architecture, changes to processor speed that may affect LocalTalk timers, and a transition to indicate change of the network cable range. At the end of this Technical Note is a sample Transition Queue procedure in both C and Pascal which includes the known transition selectors.

In addition, there is a sample Pascal source for determining whether the LAP Manager version 53 or greater exists. Calling LAPAddATQ for AppleTalk versions 52 and prior will result in a system crash since the LAP Manager is not implemented prior to AppleTalk version 53. The Boolean function, LAPMgrExists, should be used instead of checking the low memory global LAPMgrPtr, $0B18.

Calling the AppleTalk Transition Queue

System 7.0 requires the use of the MPW version 3.2 interface files and libraries. The AppleTalk interface presents two new routines for calling all processes in the AppleTalk Transition Queue. Rather than use parameter block control calls as described in Technical Note #250, "AppleTalk Phase 2," use the ATEvent procedure or the ATPreFlightEvent function to send transition notification to all queue elements. These procedures are discussed in *Inside Macintosh* Volume VI, Chapter 32.

> **Note**: You can call the ATEvent and ATPreFlightEvent routines only at virtual-memory safe time. See the Memory Management chapter of *Inside Macintosh* Volume VI, Chapter 28, for information on virtual memory.

Standard AppleTalk Transition Constants

You should use the following constants for the standard AppleTalk transitions:

```
CONST    ATTransOpen                 = 0;  {open transition }
         ATTransClose                = 2;  {prepare-to-close transition }
         ATTransClosePrep            = 3;  {permission-to-close transition }
         ATTransCancelClose          = 4;  {cancel-close transition }
         ATTransNetworkTransition    = 5;  {.MPP Network ADEV Transition }
         ATTransNameChangeTellTask   = 6;  {change-Flagship-name transition }
         ATTransNameChangeAskTask    = 7;  {permission-to-change-Flagship-name transition }
         ATTransCancelNameChange     = 8;  {cancel-change-Flagship-name transition }
         ATTransCableChange          = 'rnge' {cable range change transition }
         ATTransSpeedChange          = 'sped' {change in cpu speed }
```

The following information concerns the new transitions from `ATTransNetworkTransition` through `ATTransSpeedChange`.

The Flagship Naming Service

System 7.0 allows the user to enter a personalized name by which their system will be published when connected to an AppleTalk network. The System 'STR ' resource ID –16413 is used to hold this name. The name (listed as Macintosh Name) can be up to 31 characters in length and can be set using the File Sharing Control Panel Device. This resource is different from the Chooser name, System 'STR ' resource ID –16096. When providing network services for a workstation, the Flagship name should be used so that the user can personalize their workstation name while maintaining the use of the Chooser name for server connection identification. It's important to note that the Flagship name resource is available only from System 7.0. **DTS recommends that applications do not change either of these 'STR ' resources.**

Applications taking advantage of this feature should implement an AppleTalk transition queue to stay informed as to changes to this name. Three new transitions have been defined to communicate Flagship name changes between applications and other resident processes. Support for the Flagship Naming Service Transitions is provided starting from AppleTalk version 56. Note that AppleTalk version 56 can be installed on pre-7.0 systems; however, the Flagship Naming Service is available only from System 7.0 and later.

The ATTransNameChangeAskTask Transition

From Assembly language, the stack upon calling looks as follows:

```
ATQEvent             RECORD   0
ReturnAddr           DS.L     1       ; address of caller
```

```
theEvent             DS.L     1        ; = 7; ID of ATTransNameChangeAskTask transaction
aqe                  DS.L     1        ; pointer to task record
infoPtr              DS.L     1        ; pointer to NameChangeInfo parameter block
                     ENDR
```

The `NameChangeInfo` record block is as follows:

```
NameChangeInfoPtr: ^NameChangeInfo;
NameChangeInfo      = RECORD
                    newObjStr:       Str32;        {new Flagship name to change to }
                    name             StringPtr;    {ptr to location to place ptr to process }
                                                   {name }
                    END;
```

The `ATTransChangeNameAskTask` is issued under System 7.0 to inform Flagship clients that a process wants to change the Flagship name. Each AppleTalk Transition Queue element that processes the `ATTransChangeNameAskTask` can inspect the `NameChangeInfoPtr^.newObjStr` to determine the new Flagship name. If you deny the request, you must set the `NameChangeInfoPtr^.name` pointer with a pointer to a Pascal String buffer containing the name of your application **or** to the nil pointer. The AppleTalk Transition Queue process returns this pointer. The requesting application can display a dialog notifying the user of the name of the application that refused the process.

While processing this event, you may make synchronous calls to the Name Binding Protocol (NBP) to attempt to register your entity under the new name. It is recommended that you register an entity using the new Flagship name while handling the `ATTransChangeNameAskTask` event. You should not deregister an older entity at this point. Your routine must return a function result of 0 in the D0 register, indicating that it accepts the request to change the Flagship name, or a nonzero value, indicating that it denies the request.

DTS does not recommended that you change the Flagship name. The Sharing Setup CDEV does not handle this event and the Macintosh name will not be updated to reflect this change if the CDEV is open.

The ATTransNameChangeTellTask Transition

From Assembly language, the stack upon calling looks as follows:

```
ATQEvent             RECORD   0
ReturnAddr           DS.L     1        ; address of caller
theEvent             DS.L     1        ; = 6; ID of ATTransNameChangeTellTask transaction
aqe                  DS.L     1        ; pointer to task record
infoPtr              DS.L     1        ; pointer to the new Flagship name
                     ENDR
```

A process uses ATEvent to send the `ATTransNameChangeTellTask` to notify AppleTalk Transition Queue clients that the Flagship name is being changed. The LAP Manager then calls every routine in the AppleTalk Transition Queue that the Flagship name is being changed.

When the AppleTalk Manager calls your routine with a `ATTransNameChangeTellTask` transition, the third item on the stack is a pointer to a Pascal string of the new Flagship name to be registered. Your process should deregister any entities under the old Flagship name at this time. You may make synchronous calls to NBP to deregister an entity. Return a result of 0 in the D0 register.

> **Note**: When the AppleTalk Manager calls your process with a TellTask transition (that is, with a routine selector of `ATTransNameChangeTellTask`), you cannot prevent the Flagship name from being changed.

To send notification that your process intends to change the Flagship name, use the ATEvent function described above. Pass ATTransNameChangeTellTask as the event parameter and a pointer to the new Flagship name (Pascal string) as the infoPtr parameter.

The ATTransCancelNameChange Transition

From Assembly language, the stack upon calling looks as follows:

```
ATQEvent              RECORD   0
ReturnAddr            DS.L     1       ; address of caller
theEvent              DS.L     1       ; = 8; ID of ATTransCancelNameChange transaction
aqe                   DS.L     1       ; pointer to task record
                      ENDR
```

The ATTransCancelNameChange transition complements the ATTransNameChangeAskTask transition. Processes that acknowledged an ATTransNameChangeAskTask transition will be sent the ATTransCancelNameChange transition if a later process disallows the change of Flagship name. Your process should deregister any NBP entities registered during the ATTransNameChangeAskTask transition. You may make synchronous calls to NBP to deregister an entity. Return a result of 0 in the D0 register.

System 7.0 Sharing Setup cdev and Flagship Naming Service Interaction

The Flagship Naming Service is a new system service built into System 7. It is used to publish the workstation using the Flagship name. The Flagship Naming Service implements an AppleTalk Transition Queue element to respond to changes in the Flagship name. For example, the Sharing Setup cdev can be used to reset the Flagship name. When a new Macintosh (Flagship) name is entered in Sharing Setup, Sharing Setup sends an ATTransNameChangeAskTask message to the AppleTalk Transition Queue to request permission to change the Flagship name. The Flagship Naming Service receives the ATTransNameChangeAskTask transition and registers the new name under the type “Workstation” on the local network. Sharing Setup follows with the ATTransNameChangeTellTask to notify AppleTalk Transition Queue clients that a change in Flagship name will occur. The Flagship Naming Service responds by deregistering the workstation under the old Flagship name.

If an error occurs from the NBPRegister call, Flagship Naming Service returns a nonzero error (the error returned from NBPRegister) and a pointer to its name in the NameChangeInfoPtr^.Name field. Note that the Workstation name is still registered under the previous Flagship name at this point.

AppleTalk Remote Access Network Transition Event

AppleTalk Remote Access allows you to establish an AppleTalk connection between two Macintosh computers over standard telephone lines. If the Macintosh you dial-in to is on an AppleTalk network, such as LocalTalk or Ethernet, your Macintosh becomes, effectively, a node on that network. You are then able to use all the services on the new network. Given this new capability, it is important that services running on your Macintosh are notified when new AppleTalk connections are established and broken. For this reason, the ATTransNetworkTransition event has been added to AppleTalk version 57. This event can be expected under System 6.0.4 and greater.

Internally, both the AppleTalk Session Protocol (ASP) and the AppleTalk Data Stream Protocol (ADSP) have been modified to respond to this transition event. When a disconnect transition event is detected, these drivers close down sessions on the remote side of the connection.

The ATTransNetworkTransition Transition

From Assembly language, the stack upon calling looks as follows:

```
ATQEvent            RECORD   0
ReturnAddr          DS.L     1       ; address of caller
theEvent            DS.L     1       ; = 5; ID of ATTransNetworkTransition
aqe                 DS.L     1       ; pointer to task record
infoPtr             DS.L     1       ; pointer to the TNetworkTransition record
                    ENDR
```

The `TNetworkTransition` record block is passed as follows:

```
TNetworkTransition  RECORD   0
private             DS.L     1       ; pointer used internally by AppleTalk Remote Access
netValidProc        DS.L     1       ; pointer to the network validate procedure
newConnectivity     DS.B     1       ; true = new connectivity, false = loss of connectivity
                    ENDR
```

Network Transition Event for AppleTalk Remote Access

Network transition events are generated by AppleTalk Remote Access to inform AppleTalk Transition Queue applications and resident processes that network connectivity has changed. The type of change is indicated by the `NetTransPtr^.newConnectivity` flag. If this flag is true, a connection to a new internet has taken place. In this case, all network addresses will be returned as reachable. If the `newConnectivity` flag is false, certain networks are no longer reachable. Since AppleTalk Remote Access is connection based, it has knowledge of where a specific network exists. AppleTalk Remote Access can take advantage of that knowledge during a disconnect to inform AppleTalk Transition Queue clients that a network is no longer reachable. This information can be used by clients to age out connections immediately rather than waiting a potentially long period of time before discovering that the other end is no longer responding.

When AppleTalk Remote Access is disconnecting, it passes a network validation hook in the TNetworkTransition record, `NetTransPtr^.netValidProc`. A client can use the validation hook to ask AppleTalk Remote Access whether a specific network is still reachable. If the network is still reachable, the validate function will return true. A client can then continue to check other networks of interest until the status of each one has been determined. After a client is finished checking networks, control returns to AppleTalk Remote Access where the next AppleTalk Transition Queue client is called.

The information the network validate hook returns is valid only if a client has just been called as a result of a transition. A client can validate networks only when it has been called to handle a Network Transition Event. Note that the Network Transition Event can be called as the result of an interrupt, so a client should obey all of the normal conventions involved with being called at this time (for example, don't make calls that move memory and don't make *synchronous* Preferred AppleTalk calls).

To check a network number for validity the client uses the network validate procedure to call AppleTalk Remote Access. This call is defined using C calling conventions as follows:

```
pascal long netValidProc(TNetworkTransition *thetrans, unsigned long theAddress);

    thetrans     -->  pass in the TNetworkTransition record given to you when your
```

```
                    transition handler was called.

theAddress    -->   this is the network address you want checked. The format of
                    theAddress is the same as AddrBlock as defined in Inside
                    Macintosh II, page 281:

                    Bytes 2 & 3 (High Word) - Network Number
                    Byte 1 - Node Number
                    Byte 0 (Low Byte) - Socket Number
```

Result codes	true	network is still reachable
	false	network is no longer reachable

AppleTalk Transition Queue handlers written in Pascal must implement glue code to use the netValidProc.

Cable Range Change Transition Event

The Cable Range Transition, `ATTransCableChange,` event informs AppleTalk Transition Queue processes that the cable range for the current network has changed. This can occur when a router is first seen, when the last router ages out, or when an RTMP broadcast packet is first received with a cable range that is different from the current range. The `ATTransCableChange` event is implemented beginning with AppleTalk version 57. Most applications should have no need to process this event. This event can be expected under System 6.0.4 and greater.

The ATTransCableChange Transition

From Assembly language, the stack upon calling looks as follows:

```
ATQEvent            RECORD   0
ReturnAddr          DS.L     1        ; address of caller
theEvent            DS.L     1        ; = 'rnge'; ID of ATTransCableChange
aqe                 DS.L     1        ; pointer to task record
infoPtr             DS.L     1        ; pointer to the TNetworkTransition record
                    ENDR
```

The `TNewCRTrans` record block is passed as follows:

```
TNewCRTrans         RECORD   0
newCableLo          DS.W     1        ; the new Cable Lo received from RTMP
newCableHi          DS.W     1        ; the new Cable Hi received from RTMP
                    ENDR
```

The cable range is a range of network numbers starting from the lowest network number through the highest network number defined by a seed router for a network. All node addresses acquired on a network must have a network number within the defined cable range. For non-extended networks, the lowest and the highest network numbers are the same. If the cable range on the network changes, for example, if the router on the network goes down, the Cable Range Change event will be issued with the parameters described earlier in this Technical Note.

After receiving the event, a multinode application should use the new cable range to check if all the multinodes it obtained prior to the event are still valid. For the invalid multinodes, the application should issue the .MPP `RemoveNode` control call to get rid of invalid nodes. The .MPP `AddNode` control call can be issued immediately after removing invalid nodes to obtain new valid multinodes in the new cable range. This cable range change transition event will be issued only during system task time.

The Speed Change Transition Event

The `ATTransSpeedChange` transition event is defined for applications that change CPU speed without rebooting, to notify time dependent processes that such change has taken place. Such speed change occurs when altering the cache states on the 68030 or 68040 CPU's, or with third party accelerator cards which allow speed changes on the fly via a Control Panel device. Any process which alters the effective CPU speed should use the ATEvent to notify processes of this change. Issue the `ATTransSpeedChange` event **only** at `SystemTask` time! Any process which is dependent on changes to the CPU speed should watch for this event. The speed change transition event will be issued only during system task time.

One time dependent routine is LocalTalk, whose low-level timer values must be recalculated when the CPU speed changes. Note that altering the cache state on the 68030 does not affect LocalTalk, however doing so on the 68040 does affect LocalTalk timers. This event must be sent by any application that toggles caching on the 68040 processor on the fly. If the cache is toggled and LocalTalk is not notified, a loss of network connection will result if LocalTalk is the current network connection. Note that only LocalTalk implemented in AppleTalk version 57 or greater recognizes the speed change transition event. Contact Apple Software Licensing for licensing AppleTalk version 57.

Regarding LocalTalk on the Mac Plus, the timing values are hard-coded in ROM regardless of the CPU speed. Vendors of accelerators for Mac Pluses should contact DTS for information on how to make LocalTalk work for you.

The ATTransSpeedChange Transition

From Assembly language, the stack upon calling looks as follows:

```
ATQEvent            RECORD   0
ReturnAddr          DS.L     1        ; address of caller
theEvent            DS.L     1        ; = 'sped'; ID of ATTransSpeedChange
aqe                 DS.L     1        ; pointer to task record
                    ENDR
```

To notify LocalTalk that a change in processor speed has taken place, use the `ATEvent` procedure. Pass `ATTransSpeedChange` as the `event` parameter and a nil pointer as the `infoPtr` parameter. This event must be issued only at System Task time.

Sample Pascal Source to LAPMgrExists Function

It is important to check whether the LAP Manager exists before making LAP Manager calls like LAPAddATQ. The LAP Manager is implemented beginning with AppleTalk version 53. Rather than check the low memory global LAPMgrPtr, it is preferable to check for it's existence from a higher level. The following Pascal source demonstrates this technique.

```
FUNCTION GestaltAvailable: Boolean;
CONST
   _Gestalt = $A1AD;
BEGIN
    GestaltAvailable := TrapAvailable(_Gestalt);
    { TrapAvailable is documented in Inside Macintosh Volume VI, page 3-8 }
```

```
END;


FUNCTION AppleTalkVersion: Integer;
CONST
    versionRequested = 1; { version of SysEnvRec }
VAR
    refNum: INTEGER;
    world: SysEnvRec;
    attrib: LONGINT;
BEGIN
    AppleTalkVersion := 0;    { default to no AppleTalk }
    IF OpenDriver('.MPP', refNum) = noErr THEN { open the AppleTalk driver }
        IF GestaltAvailable THEN
        BEGIN
            IF (Gestalt(gestaltAppleTalkVersion, attrib) = noErr) THEN
                AppleTalkVersion := BAND(attrib, $000000FF);
        END
        ELSE    { Gestalt or gestaltAppleTalkVersion selector isn't available }
            IF SysEnvirons(versionRequested, world) = noErr THEN
                AppleTalkVersion := world.atDrvrVersNum;
END;


FUNCTION LAPMgrExists: Boolean;
BEGIN
    { AppleTalk phase 2 is in AppleTalk version 53 and greater }
    LAPMgrExists := (AppleTalkVersion >= 53);
END;
```

Sample AppleTalk Transition Queue Function

A sample AppleTalk Transition Queue function has been implemented in both C and Pascal. These samples have been submitted as snippet code to appear on the Developer CD. Since Transition Queue handlers are called with a C style stack frame, the Pascal sample includes the necessary C glue.

Sample AppleTalk Transition Queue Function in C

The following is a sample AppleTalk Transition Queue handler for C programmers. To place the handler in the AppleTalk Transition Queue, define a structure of type myATQEntry in the main body of the application. Assign the SampleTransQueue function to the myATQEntry.CallAddr field. Use the LAPAddATQ function to add the handler to the AppleTalk Transition Queue. Remember to remove the handler with the LAPRmvATQ function before quitting the application.

Warning: The System 7 Tuner extension will not load AppleTalk resources if it detects that AppleTalk is off at boot time. Remember to check the result from the `LAPAddATQ` function to determine whether the handler was installed successfully.

The following code was written with MPW C v3.2.

```
/*-------------------------------------------------------------------
  file: TransQueue.h
-------------------------------------------------------------------*/

#include <AppleTalk.h>

/*
 *  Transition queue routines are designed with C calling conventions in mind.
 *  They are passed parameters with a C style stack and return values are expected
 *  to be in register D0.
 */

#define ATTransOpen                    0       /* .MPP just opened */
#define ATTransClose                   2       /* .MPP is closing */
#define ATTransClosePrep               3       /* OK for .MPP to close? */
#define ATTransCancelClose             4       /* .MPP close was canceled */
#define ATTransNetworkTransition       5       /* .MPP Network ADEV transition */
#define ATTransNameChangeTellTask      6       /* Flagship name is changing */
#define ATTransNameChangeAskTask       7       /* OK to change Flagship name */
#define ATTransCancelNameChange        8       /* Flagship name change was canceled */
#define ATTransCableChange             'rnge'  /* Cable Range Change has occurred */
#define ATTransSpeedChange             'sped'  /* Change in processor speed has occurred */

/*-----------------------------------------------------------------------
       NBP Name Change Info record
-----------------------------------------------------------------------*/
typedef struct NameChangeInfo {
    Str32  newObjStr;     /* new NBP name */
    Ptr    name;          /* Ptr to location to place a pointer to Pascal string of */
                          /* name of process that NAK'd the event */
}
    NameChangeInfo, *NameChangePtr, **NameChangeHdl;

/*-----------------------------------------------------------------------
       Network Transition Info Record
-----------------------------------------------------------------------*/

typedef struct TNetworkTransition {
    Ptr        private;          /* pointer to private structure */
    ProcPtr    netValidProc;     /* pointer to network validation procedure */
    Boolean    newConnectivity;  /* true = new connection, */
                                 /* false = loss of connection */

}
    TNetworkTransition , *TNetworkTransitionPtr, **TNetworkTransitionHdl;

typedef    pascal long    (*NetworkTransitionProcPtr)(TNetworkTransitionPtr netTrans, \
                      unsigned long theNet);

/*-----------------------------------------------------------------------
       Cable Range Transition Info Record
-----------------------------------------------------------------------*/
typedef struct TNewCRTrans {
    short  newCableLo;    /* the new Cable Lo received from RTMP */
    short  newCableHi;    /* the new Cable Hi received from RTMP */
}
    TNewCRTrans , *TNewCRTransPtr, **TNewCRTransHdl;

/*-----------------------------------------------------------------------
       AppleTalk Transition Queue Element
-----------------------------------------------------------------------*/
typedef struct    myATQEntry {
    Ptr        qLink;     /* -> next queue element */
    short      qType;     /* unused */
    ProcPtr    CallAddr;  /* -> transition procedure */
    Ptr        globs;     /* -> to user defined globals */
}
    myATQEntry, *myATQEntryPtr, **myATQEntryHdl;


/*-----------------------------------------------------------------------
  file: TransQueue.c
-----------------------------------------------------------------------*/

#include <Memory.h>
```

```
#include <AppleTalk.h>
#include "TransQueue.h"

long SampleTransQueue(long selector, myATQEntry *q, void *p)
{
    long                        returnVal = 0; /* return 0 for unrecognized events */
    NameChangePtr               myNameChangePtr;
    TNewCRTransPtr              myTNewCRTransPtr;
    TNetworkTransitionPtr       myTNetworkTransitionPtr;
    NetworkTransitionProcPtr    myNTProcPtr;
    StringPtr                   newNamePtr;
    long                        checkThisNet;
    char                        **t;
    short                       myCableLo, myCableHi;


    /*
     * This is the dispatch part of the routine. We'll check the selector passed into
     * the task; its location is 4 bytes off the stack (selector).
     */
    switch(selector) {
        case ATTransOpen:
            /*
             *  Someone has opened the .MPP driver. This is where one would reset the
             *  application to its usual network state (i.e., you could register your
             *  NBP name here). Always return 0.
             */
            break;

        case ATTransClose:
            /*
             *  When this routine is called, .MPP is going to shut down no matter what we
             *  do. Handle that type of situation here (i.e., one could remove an NBP
             *  name and close down all sessions); 'p' will be nil. Return 0
             *  to indicate no error.
             */
            break;

        case ATTransClosePrep:
            /*
             *  This event gives us the chance to deny the closing of AppleTalk if we
             *  want. Returning   a value of 0 means it's OK to close; nonzero
             *  indicates we'd rather not close at this time.
             *
             *  With this event, the parameter 'p' actually means something. 'p' in
             *  this event is a pointer to an address that can hold a pointer to a
             *  string of our choosing. This string indicates to the user which task
             *  would rather not close. If you don't want AppleTalk to close, but you
             *  don't have a name to stick in there, you MUST place a nil value in
             *  there instead.
             *
             *   (We're doing this all locally to this case because it's C and we can, so
             *  there.)
             */
            newNamePtr = (StringPtr)NewPtr(sizeof(Str32));

            /*
             *  Assume Ptr allocation successful.
             */

            newNamePtr = "\pBanana Mail";    /* this will either be an Ax reference or PC
             *  relative depending on compiler and options */

            /*
             *  get a new reference to the address we were passed (in a form we can use)
             */
        t = (char **) p;
        /*
         *  place the address of our string into the address we were passed
         */
        *t = (char *)newNamePtr;

        /*
         *  return a nonzero value so that AppleTalk knows we'd rather not close
         */
        returnVal = 1;
        break;

    case ATTransCancelClose:
        /*
         *  Just kidding, we didn't really want to cancel that AppleTalk closing
         *  after all. Reset all your network activities that you have disabled
         *  here (if any). In our case, we'll just fall through. 'p' will be nil.
         */
        break;

    case ATTransNetworkTransition:
        /*
         *  A Remote AppleTalk connection has been made or broken.
         *  'p' is a pointer to a TNetworkTransition record.
         *   Always return 0.
         */
        myTNetworkTransitionPtr = (TNetworkTransitionPtr)p;
        /*
         *  Check newConnectivity element to determine whether
         *  Remote Access is coming up or going down
         */
        if (myTNetworkTransitionPtr->newConnectivity) {
            /*
             * Have a new connection
             */
        }
        else {
            /*
             * Determine which network addresses need to be validated
             * and assign the value to checkThisNet.
             */
            checkThisNet = 0x1234FD80;  /* network 0x1234, node 0xFD, socket 0x80 */
            myNTProcPtr = (NetworkTransitionProcPtr)myTNetworkTransitionPtr->netValidProc;
            if ((*myNTProcPtr)(myTNetworkTransitionPtr, checkThisNet)) {
                /*
                 * Network is still valid
                 */
            }
            else {
                /*
                 * Network is no longer valid
                 */
            }
        }
        break;

    case ATTransNameChangeTellTask:
        /*
         *  Someone is changing the Flagship name and there is nothing we can do.
         *  The parameter 'p' is a pointer to a Pascal style string which holds new
         *  Flagship name.
         */
        newNamePtr = (StringPtr) p;

        /*
         *  You should deregister any previously registered NBP entries under the
```

```
             *  'old' Flagship name. Always return 0.
             */
            break;

        case ATTransNameChangeAskTask:
            /*
             *  Someone is messing with the Flagship name.
             *  With this event, the parameter 'p' actually means something. 'p' is
             *  a pointer to a NameChangeInfo record. The newObjStr field contains the
             *  new Flagship name. Try to register a new entity using the new Flagship name.
             *  Returning a value of 0 means it's OK to change the Flagship name.
             */
            myNameChangePtr = (NameChangePtr)p;

            /*
             *  If the NBPRegister is unsuccessful, return the error. You must also set
             *  p->name pointer with a pointer to a Pascal style string of the process
             *  name.
             */
            break;

        case ATTransCancelNameChange:
            /*
             *  Just kidding, we didn't really want to change that name after
             *  all. Remove new NBP entry registered under the ATTransNameChangeAskTask
             *  Transition. In our case,  we'll just fall through. 'p' will be nil. Remember
             *  to return 0.
             */
            break;

        case ATTransCableChange:
            /*
             *  The cable range for the network has changed. The pointer 'p' points
             *  to a structure with the new network range. (TNewCRTransPtr)p->newCableLo
             *  is the lowest value of the new network range. (TNewCRTransPtr)p->newCableHi
             *  is the highest value of the new network range. After handling this event,
             *  always return 0.
             */
            myTNewCRTransPtr = (TNewCRTransPtr)p;
            myCableLo = myTNewCRTransPtr->newCableLo;
            myCableHi = myTNewCRTransPtr->newCableHi;
            break;

        case ATTransSpeedChange:
            /*
             *  The processor speed has changed. Only LocalTalk responds to this event.
             *  We demonstrate this event for completeness only.
             *  Always return 0.
             */
            break;

        default:
            /*
             *  For future transition queue events. (and yes, Virginia, there will be more)
             */
            break;
    } /* end of switch */

    /*
     * return value in register D0
     */
    return returnVal;
}
```

Sample AppleTalk Transition Queue Function in Pascal

The following is a sample AppleTalk Transition Queue handler for Pascal programmers. AppleTalk Transition Queue handlers are passed parameters using the C parameter passing convention. In addition, the 4 byte function result must be returned in register D0. To meet this requirement, a C procedure is used to call the handler, then to place the 4 byte result into register D0. The stub procedure listing follows the handler.

To place the handler in the AppleTalk Transition Queue, define a structure of type myATQEntry in the main body of the application. Assign the CallTransQueue C procedure to the myATQEntry.CallAddr field. Use the LAPAddATQ function to add the handler to the AppleTalk Transition Queue. Remember to remove the handler with the LAPRmvATQ function before quitting the application.

Warning: The System 7 Tuner extension will not load AppleTalk resources if it detects that AppleTalk is off at boot time. Remember to check the result from the `LAPAddATQ` function to determine whether the handler was installed successfully.

The following code was written with MPW Pascal and C v3.2.

```
{********************************************************************************
  file: TransQueue.p
********************************************************************************}

UNIT TransQueue;

INTERFACE

USES MemTypes, QuickDraw, OSIntF, AppleTalk;

CONST
(*  Comment the following 4 constants since they are already defined in the AppleTalk unit
    ATTransOpen                  =  0;  { .MPP is opening }
    ATTransClose                 =  2;  { .MPP is closing }
    ATTransClosePrep             =  3;  { OK for .MPP to close? }
    ATTransCancelClose           =  4;  { .MPP close was canceled }
*)
    ATTransNetworkTransition     =  5;  { .MPP Network ADEV transition }
    ATTransNameChangeTellTask    =  6;  { Flagship name is changing }
    ATTransNameChangeAskTask     =  7;  { OK to change Flagship name }
    ATTransCancelNameChange      =  8;  { Flagship name change was canceled }
    ATTransCableChange           =  'rnge'; { Cable Range Change has occurred }
    ATTransSpeedChange           =  'sped'; { Change in processor speed has occurred }

{----------------------------------------------------------------------
       NBP Name Change Info record
----------------------------------------------------------------------}

TYPE

NameChangeInfo = RECORD
    newObjStr  : Str32;       { new NBP name }
    name       : Ptr;         { Ptr to location to place a pointer to Pascal string of }
                              { name of process that NAK'd the event }
    END;
NameChangePtr = ^NameChangeInfo;
NameChangeHdl = ^NameChangePtr;

{----------------------------------------------------------------------
       Network Transition Info Record
----------------------------------------------------------------------}
```

```
TNetworkTransition = RECORD
    private         : Ptr;          { pointer to private structure }
    netValidProc    : ProcPtr;      { pointer to network validation procedure }
    newConnectivity : Boolean;      { true = new connection, }
                                    { false = loss of connection }
    END;

TNetworkTransitionPtr = ^TNetworkTransition;
TNetworkTransitionHdl = ^TNetworkTransitionPtr;

{ The netValidProc procedure has the following C interface. Note the }
{ CallNetValidProc C function, which follows. The C Glue routine allows the Pascal }
{ handler to make the call to the netValidProc function. }

{
typedef pascal long   (*NetworkTransitionProcPtr)(TNetworkTransitionPtr netTrans, \
                    unsigned long theNet);
}
{----------------------------------------------------------------------
       Cable Range Transition Info Record
----------------------------------------------------------------------}
TNewCRTrans = RECORD
    newCableLo     : INTEGER;     { the new Cable Lo received from RTMP }
    newCableHi     : INTEGER;     { the new Cable Hi received from RTMP }
    END;
TNewCRTransPtr = ^TNewCRTrans;
TNewCRTransHdl = ^TNewCRTransPtr;

{----------------------------------------------------------------------
       AppleTalk Transition Queue Element
----------------------------------------------------------------------}
myATQEntry = RECORD
    qlink      : Ptr;          { -> next queue element }
    qType      : INTEGER;      { unused }
    CallAddr   : ProcPtr;      { -> transition procedure }
    globs      : Ptr;          { -> to user defined globals }
    END;
myATQEntryPtr = ^myATQEntry;
myATQEntryHdl = ^myATQEntryPtr;


{---------------- Prototypes --------------------}

FUNCTION SampleTransQueue (selector :LONGINT; q :myATQEntryPtr;  p :Ptr) : LONGINT;
{
 *  Transition queue routines are designed with C calling conventions in mind.
 *  They are passed parameters with a C style stack and return values are expected
 *  to be in register D0. Note that the CallTransQueue C glue routine is used
 *  to reverse the C style stack to Pascal style before calling the handler. The
 *  procedure CallTransQueue follows this listing. To install this Trans Queue
 *  handler, assign CallTransQueue to the CallAddr field, NOT SampleTransQueue.
 }

FUNCTION CallNetValidProc(p : ProcPtr; netTrans : TNetworkTransitionPtr;
                          theNet : LONGINT) : LONGINT;
{
 *  CallNetValidProc is used to call the netValidProc passed in the TNetworkTransition
 *  record. Since Pascal cannot call the ProcPtr directly, a C glue routine is
 *  used. This routine is defined following this listing.
 }


IMPLEMENTATION

FUNCTION SampleTransQueue (selector :LONGINT; q :myATQEntryPtr;  p :Ptr) : LONGINT;
```

```
VAR
    returnVal                   : LONGINT;
    myNameChgPtr                : NameChangePtr;
    myTNewCRTransPtr            : TNewCRTransPtr;
    myTNetworkTransitionPtr     : TNetworkTransitionPtr;
    newNamePtr                  : StringPtr;
    processNameHdl              : StringHandle;
    myCableLo, myCableHi        : INTEGER;
    shortSelector               : INTEGER;
    checkThisNet                : LONGINT;


BEGIN
    returnVal := 0;                     { return 0 for unrecognized events }
    {
     *  This is the dispatch part of the routine. We'll check the selector passed into
     *  the task; its location is 4 bytes off the stack (selector).
     }
    IF ((selector <= ATTransCancelNameChange) AND (selector >= ATTransOpen)) THEN
    {
     *  Check whether a numeric selector is being used whose known values are between
     *  8 and 0 so that we can implement a CASE statement with an INTEGER var.
     }
    BEGIN
        shortSelector := selector;
        CASE shortSelector OF
            ATTransOpen:
            BEGIN
                {
                 *  Someone has opened the .MPP driver. This is where one would reset the
                 *  application to its usual network state. (i.e., you could register your
                 *  NBP name here). Always return 0.
                 }
            END;

            ATTransClose:
            BEGIN
                {
                 *  When this routine is called .MPP is going to shut down no matter what we
                 *  do. Handle that type of situation here (i.e., one could remove an NBP
                 *  name and close down all sessions). 'p' will be nil. Return 0 to
                 *  indicate no error.
                 }
            END;

            ATTransClosePrep:
            BEGIN
                {
                 *  This event gives us the chance to deny the closing of AppleTalk IF we
                 *  want. Returning a value of 0 means it's OK to close; nonzero
                 *  indicates we'd rather not close at this time.
                 *
                 *  With this event, the parameter 'p' actually means something. 'p' in
                 *  this event is a pointer to an address which can hold a pointer to a
                 *  string of our choosing. This string indicates to the user which task
                 *  would rather not close. If you don't want AppleTalk to close, but you
                 *  don't have a name to stick in there,  you MUST place a nil value in
                 *  there instead.
                 }

                {
                 *  Get a new reference to the address we were passed (in a form we can use)
                 *  (We're doing this all locally to this case because we can, so
                 *  there.)
                 }
                processNameHdl := StringHandle(NewHandle(sizeof(Str32)));
    {
     *  place the address of our string into the address we were passed
     }
     := 'Banana Mail';
    Ptr(p) := Ptr(processNameHdl);

    {
     *  return a nonzero value so AppleTalk knows we'd rather not close
     }
    returnVal := 1;
END;

ATTransCancelClose:
BEGIN
    {
     *  Just kidding, we didn't really want to cancel that AppleTalk closing
     *  after all. Reset all your network activities that you have disabled here
     *  (IF any). In our case, we'll just fall through. 'p' will be nil.
     }
END;

ATTransNetworkTransition:
BEGIN
    {
     *  A Remote AppleTalk connection has been made or broken.
     *  'p' is a pointer to a TNetworkTransition record.
     *   Always return 0.
     }
    myTNetworkTransitionPtr := TNetworkTransitionPtr(p);
    {
     *  Check newConnectivity element to determine whether
     *  Remote Access is coming up or going down
     }
    if (myTNetworkTransitionPtr^.newConnectivity) THEN
    BEGIN
        {
         * Have a new connection
         }
    END
    ELSE
    BEGIN
        {
         * Determine which network addresses need to be validated
         * and assign the value to checkThisNet.
         }
        checkThisNet = $1234FD80;  /* network $1234, node $FD, socket $80 */
        if (CallNetValidProc(myTNetworkTransitionPtr^.netValidProc,
                myTNetworkTransitionPtr, checkThisNet) <> 0) THEN
        BEGIN
            {
             * Network is still valid
             }
        END
        ELSE
        BEGIN
            {
             * Network is no longer valid
             }
        END;
    END;
END;

ATTransNameChangeTellTask:
BEGIN
    {
```

```
             *  Someone is changing the Flagship name and there is nothing we can do.
             *  The parameter 'p' is a pointer to a Pascal style string which holds new
             *  Flagship name.
            }
            newNamePtr := StringPtr (p);

            {
             *  You should deregister any previously registered NBP entries under the
             *  'old' Flagship name. Always return 0.
             }
        END;

        ATTransNameChangeAskTask:
        BEGIN
            {
             *  Someone is messing with the Flagship name.
             *  With this event, the parameter 'p' actually means something. 'p' is
             *  a pointer to a NameChangeInfo record. The newObjStr field contains the
             *  new Flagship name. Try to register a new entity using the new Flagship
             *  name. Returning a value of 0 means it's OK to change the Flagship name.
             }
            myNameChgPtr := NameChangePtr (p);

            {
             *  If the NBPRegister is unsuccessful, return the error. You must also set
             *  p->name pointer with a pointer to a string of the process name.
             }
        END;

        ATTransCancelNameChange:
        BEGIN
            {
             *  Just kidding, we didn't really want to cancel that name change after
             *  all. Remove new NBP entry registered under the
             *  ATTransNameChangeAskTask Transition. 'p' will be nil.
             *  Remember to return 0.
             }
        END;

        OTHERWISE
            returnVal := 0;
            {
             *  Just in case some other numeric selector is implemented.
             }
    END; { CASE }
END
ELSE IF (ResType(selector) = ATTransCableChange) THEN
BEGIN
    {
     *  The cable range for the network has changed. The pointer 'p' points
     *  to a structure with the new network range. (TNewCRTransPtr)p->newCableLo
     *  is the lowest value of the new network range. (TNewCRTransPtr)p->newCableHi
     *  is the highest value of the new network range. After handling this event,
     *  always return 0.
     }
    myTNewCRTransPtr := TNewCRTransPtr(p);
    myCableLo := myTNewCRTransPtr^.newCableLo;
    myCableHi := myTNewCRTransPtr^.newCableHi;
    returnVal := 0;
END
ELSE IF (ResType(selector) = ATTransSpeedChange) THEN
BEGIN
    {
     *  The processor speed has changed. Only LocalTalk responds to this event.
     *  We demonstrate this event for completeness only.
     *  Always return 0.
```

```
        }
        returnVal := 0;
    END; { IF }

    SampleTransQueue := returnVal;
END;

FUNCTION CallNetValidProc(p : ProcPtr; netTrans : TNetworkTransitionPtr;
                          theNet : LONGINT) : LONGINT; EXTERNAL;

END. { of UNIT }


/********************************************************************************
  file: CGlue.c
********************************************************************************/
#include <AppleTalk.h>

/*----------------------------------------------------------------------
       Network Transition Info Record
----------------------------------------------------------------------*/

typedef struct TNetworkTransition {
    Ptr        private;          /* pointer to private structure */
    ProcPtr    netValidProc;     /* pointer to network validation procedure */
    Boolean    newConnectivity;  /* true = new connection, */
                                 /* false = loss of connection */

}
    TNetworkTransition , *TNetworkTransitionPtr, **TNetworkTransitionHdl;

typedef    pascal long    (*NetworkTransitionProcPtr)(TNetworkTransitionPtr netTrans, \
                    unsigned long theNet);

/*----------------------------------------------------------------------
       AppleTalk Transition Queue Element
----------------------------------------------------------------------*/
typedef struct    myATQEntry {
    Ptr        qLink;     /* -> next queue element */
    short      qType;     /* unused */
    ProcPtr    CallAddr;  /* -> transition procedure */
    Ptr        globs;     /* -> to user defined globals */
}
    myATQEntry, *myATQEntryPtr, **myATQEntryHdl;

/*----------------------------------------------------------------------
       Prototypes
----------------------------------------------------------------------*/
pascal long  SampleTransQueue (long selector, myATQEntry *q, void *p);
long CALLTRANSQUEUE(long selector, myATQEntry *q, void *p);
/* Capitalize CALLTRANSQUEUE so that linker can match this entry with */
/* the pascal call */
pascal long CallNetValidProc(ProcPtr p, TNetworkTransitionPtr netTrans, long theNet);


long CALLTRANSQUEUE(long selector, myATQEntry *q, void *p)
/* CallTransQueue sets up the pascal stack for the SampleTransQueue handler */
/* then puts the result into D0 */
{
    return(SampleTransQueue(selector, q, p));
}

pascal long CallNetValidProc(ProcPtr p, TNetworkTransitionPtr netTrans, long theNet)
/* CallNetValidProc is used to call the netValidProc pointed to by ProcPtr p. */
{
    NetworkTransitionProcPtr myNTProcPtr;
    myNTProcPtr = (NetworkTransitionProcPtr)p;
    return ((*myNTProcPtr)(netTrans, theNet));
}
```

Further Reference:

- *Inside AppleTalk,* Second Edition, Addison-Wesley
- *Inside Macintosh*, Volume II, The AppleTalk Manager, Addison-Wesley
- *Inside Macintosh*, Volume V, The AppleTalk Manager, Addison-Wesley
- *Inside Macintosh*, Volume VI, The AppleTalk Manager, Addison-Wesley
- *Macintosh AppleTalk Connections Programmer's Guide*, Final Draft 2, Apple Computer, Inc. (M7056/A)
- *AppleTalk Phase 2 Protocol Specification*, Apple Computer, Inc. (C0144LL/A)
- Macintosh Portable Developer Notes (DTS)
- *AppleTalk Remote Access Developer's Toolkit,* Apple Computer, Inc. (R0128LL/A)
- Technical Note #250, AppleTalk Phase 2, (DTS)
- *Alternate AppleTalk for System 7.0*, (DTS)

#312: Fun With PrJobMerge

Revised by: Matt Deatherage May 1992
Written by: Scott "Zz" Zimmerman and Matt Deatherage March 1992

This Technical Note discusses some interesting behavior you'll encounter while using PrJobMerge with the 7.0 and 7.1 versions of the LaserWriter driver.

Changes since March 1992: Corrected the Vulcan-like "THPring" typo to correctly read "THPrint," and changed a comment in the code to mean what I originally meant.

Like many Printing Manager calls, `PrJobMerge` is implemented by the currently chosen printer driver. This makes sense after consideration—since the printer driver may store job-specific information anywhere in the print record, only the printer driver can correctly merge this into a destination print record.

The LaserWriter driver's implementation of `PrJobMerge` has a few bugs in versions 7.0 and 7.1.

Fun Thing #1

Historically, `PrJobMerge` hasn't worked correctly in the LaserWriter driver. The driver does not correctly merge all job-related data (like the number of copies requested) into the destination print record, so printing multiple copies of multiple documents from the Finder isn't really possible with the LaserWriter driver.

The only possible workaround is to present a different job dialog for each document to be printed, but this isn't recommended—especially since the job dialog doesn't tell you which document you're about to print.

Fun Thing #2

As if this wasn't enough excitement for one driver, in versions 7.0 and 7.1 of the LaserWriter driver `PrJobMerge` actually manages to destroy all the job-specific information in the *source* print record after it doesn't copy it into the destination print record.

There is a workaround for this problem—make a copy of the source print record and pass the copy to `PrJobMerge`. If you pass the copy to `PrJobMerge`, you can just replace `PrJobMerge` with your own routine that makes a copy, merges it into the destination, and disposes of the copy. This will work for all printer drivers, although it's necessary only with version 7.0 of the LaserWriter driver.

Such a procedure might look like this in Pascal:

```
PROCEDURE NewPrJobMerge(hPrintSrc,hPrintDst: THPrint);

    VAR
        copyError: OSErr;
        hPrintTemp: THPrint;

    BEGIN
        hPrintTemp := hPrintSrc;  {make our own copy of the print record handle}
        copyError := HandToHand(Handle(hPrintTemp));
        PrSetError(copyError); {so we can get it later}
        IF copyError = noErr THEN BEGIN
            {hPrintTemp is now a copy of the original source record}
            PrJobMerge(hPrintTemp,hPrintDst); {This messes up hPrintTemp, but we don't care}
        END; {if copyError = noErr}
        IF hPrintTemp <> NIL THEN DisposHandle(Handle(hPrintTemp)); {only a copy, remember!}
    END;
```

Don't Go Overboard Trying to Solve This

Although the bugs in `PrJobMerge` in versions 7.0 and 7.1 of the LaserWriter driver make certain kinds of printing multiple documents impossible without device-specific workarounds, we strongly encourage you **not** to implement such code. Any code that tries to replicate the function of `PrJobMerge` must by nature depend on how the LaserWriter driver stores information in the print record, and this is a Bad Thing. The road to Compatibility Hell is paved with good intentions.

If you write your code as described in this Note, it will behave properly when the bug is fixed without change on your part. If you go overboard trying to write your own `PrJobMerge` function, your application is a prime candidate for compatibility problems.

Further Reference:

- *Inside Macintosh*, Volume II, The Printing Manager

Macintosh
Technical Notes

Developer Technical Support

#313: Performance Tuning with Development Tools

Written by: Kent Sandvik

May 1992

This Technical Note is a collection of useful ideas and suggestions to help you decrease the time required to compile and link under MPW. Some of the issues are even relevant to any development tools running under the Macintosh environment. The Tech Note will also clarify what performance tunings work, and which are marginal or may not work at all.

Introduction

This Technical Note contains information that will help you improve both compilation and linking times, and also point out about performance tricks that are marginal, or may not work at all. Most likely this information will be updated and modified as we gain more knowledge of how to speed up compilations and link stages. This Note is biased towards the `MPW` environment. However, there are many ideas that can be used with any other Macintosh development platforms. The issues are ordered from hardware- or system-related issues to specific MPW and MPW tool issues, and these are not listed in any particular order of efficiency.

Many of these ideas are benchmarked and the results are marked with a special note at the end of the paragraph . The equipment/environment consisted of:

- *Macintosh 900 Quadra, 160Mb internal hard disk, 20Mb physical RAM memory, no VM (unless stated)*
- *System 7.0.1 + TuneUp*
- *MPW 3.2 4Mb application heap, 256k file cache*
- *No extensions loaded, no network*
- *MacApp 3.0 Calc application source, C++ (when applicable)*

Hardware Issues

Accelerator Cards

We at DTS have had mixed input about using CPU accelerator cards. In general they speed up number-crunching. However, they don't help with file I/O bottleneck situations. Also they can cause compatibility problems with the tools, so we strongly recommend taking out the card for testing if you encounter weird problems during compilation or linking.

Before you buy a card, contact the appropriate accelerator card tech support group, and ask about the compatibility of their card with various Macintosh development systems. Try to borrow a card for a quick test to figure out if buying the card is justified or not and if it works with the particular development tools needed. Remember that accelerator cards based on NuBus™ can easily congest the NuBus bus (which has a 10 MB/sec limit on data transfers). Any possible savings in CPU execution could be lost in the NuBus transfers.

SCSI I/O Cards and Hard Disk Access Times

File I/O is one of the known bottlenecks that affect MPW performance (however, it's not the only bottleneck). Faster NuBus SCSI cards (like SCSI-2 cards) should certainly improve the file I/O; how much depends on the file I/O access figures. Also, shop around for hard disks with fast access times.

System Issues

Background Processes

Every background process, including the Finder, takes CPU cycles that could be used for compilation and linking. Try to limit the background tasks on your compiling system. If possible turn off any unnecessary inits and drivers.

External Sources of Interrupts

You should be aware that the development machine connected to a network will receive outside interrupts as part of the network protocol handling (as in AppleTalk), and this will also consume needed CPU cycles. For example, System 7 Personal FileShare requires a lot of attention from the system itself. This is also true of any other background communication protocol handling. The best possible case is a standalone development system. However, for practical reasons (like accessing common volumes) a developer can seldom afford to configure a standalone environment. If possible, minimize the network access on the development system.

If a server is connected to the system and is not used, the Finder still attempts to keep its desktop in synch with the server. This consumes CPU cycles. If the server and the network are busy then the machine is stuck waiting. If possible, always remove servers from the desktop when they are no longer needed.

> Δ System in network + inits† = 404s, no-network System = 379s, savings 6.25%.
> † mail services, 4 servers on desktop, 1 application in memory

File Cache

We suggest that you experiment with your particular system configuration, there is no magic value which we could recommend. There's however a difference between System 6.0.x and 7.x. Large cache sizes in System 6 will not improve the performance due to a bug, which is fixed in System 7. If your development tool is mostly in memory during the execution, many of the resources and data files may be cached in memory between the execution stages. Following is a test where we changed the cache size between 32k and 4096k while compiling the same sources. As shown we didn't directly find an optimal value, so the 96k value is a good approximation.

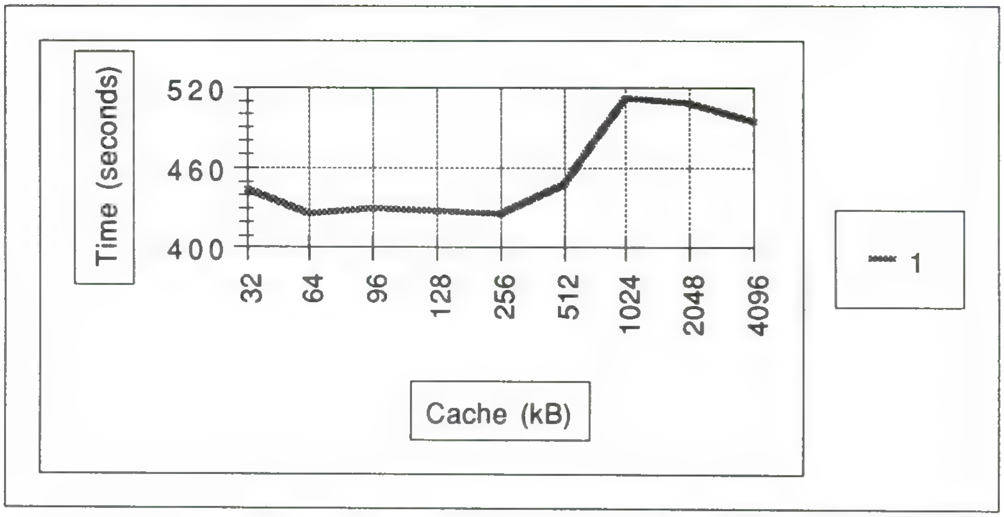


Figure 1 - cache size vs. compilation time

System 7 Virtual Memory

The use of virtual memory is recommended when you would like to have many development tools running at the same time. However, VM is much slower than real memory; it constantly needs to read/write to the much slower hard disk. One exception is the IIci with its built-in video. IIci has a non-contiguous memory map, and uses the MMU to map the logical addresses. The algorithms used by the ROM are slower thant the ones used by VM. However the access gtes slower if a page fault occurs.

Note also that 32-bit mode runs faster than 24-bit mode.

Δ VM on† = 481s, VM off = 400s, savings 16.8%.
†12 Mb VM

Compiler Issues

Compiler Flags

Eliminate any compiler flags that are not necessary for the code compilation. For example, optimization flags take more CPU cycles, and in many cases the code produced without optimization is OK for a quick syntax or functionality test. Read the manuals carefully; they should indicate which flags are default ones. Note also that the MacApp has its own `Startup` file where many compiler flags are defined.

The `-sym on/full` flag will trigger `.SYM` file information generation, and this takes time. If possible avoid symbolics generation until you really need to debug.

Selected #include File Handling

A larger percent of a typical compile is spent reading the header files, so reading them only once for each source file compilation is a win. We are talking about cases where various source files each want to include the same header file. The MPW C compiler has a special pragma called #pragma once, which will make sure that each source file with this statement at the top of the file will be read in only once. However this will not work with other languages - like C++ - so the following guidelines are useful:

In your individual source files, bracket your C++ include files so that they are not read more than once during a compilation of a source file:

```
//Utilities.cp

#ifndef _UTILITIES_
#include "Utilities.h"
#endif _UTILITIES_

// code....
```

Of course, you also need to put bracketing into your local include files so that things don't go haywire if you do include the same file twice (note that we recommend using only one underline, because ANSI C has reserved the use of two underlines):

```
// Utilities.h

#ifndef _UTILITIES_
#define _UTILITIES_
// definitions

#endif _UTILITIES_
// place the line above at the end of the file
```

One trick is to define a global `IncludeFiles.h` file, which contains all files needed for the other header files, and include it on demand inside the other header files, using the `#ifndef` trick. A `#pragma once` statement placed first in a C header file provides the same functionality. However, we can't guarantee that the `#pragma once` statements in C++ code end up in the right places with the generated C code, so don't use this with C++.

Δ No #include labels = 183s, include labels = 174s, savings 6.45%.

C / C++ Compiler Issues

Load/Dump

`load/dump` is described in the MPW C++ Release Notes. It provides the biggest single performance improvement possible when using MPW C++. Use the `-load` and `-dump` flags instead of the MPW C `#pragma load/dump` statements, because they work differently. `#pragma load` and `#pragma dump` placed directly in C++ may have Cfront generate code that appears before the pragma and thus could cause the `load/dump` to work incorrectly.

MPW C also has this feature, implimented using #pragma load and #pragma dump - compile time savings are similar to those found in C++. For more information on how to use this feature, please refer to the MPW C 3.0 release notes, pages 40-41.

Tradeoffs Between Compiling Small and Large Files

Each time CPlus is triggered, MPW will load in resources needed for the compiler. This also happens when CPlus triggers the C compiler. In the case of a compilation of 10 files, the C++ and C resources are loaded 10 times in a row.

There are cases where a huge file compilation is faster than compiling a number of smaller files. The overall trick is to create dependencies (Makefiles) where as few files as possible are recompiled when something changes.

Pascal Compiler Issues

Limit Symbol Table Generation

The `-p` switch on the Pascal compiler is useful to determine where the compiler is spending its time. For example, with MacApp the compiler should spend a lot of time inside the MacApp units and in the `PInterfaces` files when discarding predigested symbol resources on the interface files and reanalyzing the source.

The goal is to to configure a standard set of interface files so that we can use the precompiled symbol information. Changing the order of includes or `USES` statements could cause these resources to be rebuilt, taking extra time.

Here is a possible strategy to help you analyze the information from the compiler and define strategies that will minimize the need for resources and make the compilation faster:

1. ALL units should use `Types`, `QuickDraw`, `Packages`, `SANE`, and `Printing` (if needed).

2. When additional units are required, *always* use units from the newer, smaller groupings:
   `Events`, `Controls`, `Desk`, `Windows`, `TextEdit`, `Dialogs`, `Fonts`, `Lists`, `Menus`, `Resources`, `Scrap`, and `ToolUtils` {instead of `ToolIntf`}
   `OSUtils`, `Files`, `Devices`, `DeskBus`, `DiskInit`, `Disks`, `Errors`, `Memory`, `OSEvents`, `Retrace`, `SegLoad`, `Serial`, `ShutDown`, `Slots`, `Sound`, `Start` and `Timer` {instead of `OSIntf`}
   and
   `Script`, `Palettes`, `Picker`, `Perf`, `DisAsmLookup`, `AppleTalk`

Note that if you mix references to newer and older files it will take longer to compile.

3. Always use units in the same order. The Apple units set compile flags that *must* be identical the next time a unit is used or the compiler will not use the symbolic resources.

4. The MacApp units also set compile flags, so they should appear after the `MPW:Interfaces:PInterfaces` units and always in the same order.

5. Keep your own units in a consistent order in each USES statement, especially if you use compiler variables in your source.

6. Adjust your build scripts to build units in the same order they are listed in your USES statements.

7. As you clean up the units, compile them in a full build with the `-p` compiler option to verify the results of your work. The output will indicate when the compiler uses the resources.

If you clean up your files in the order in which the units are built, you will begin to accumulate savings as you go along. However, don't expect to see a tremendous difference until nearly all your USES statements have been cleaned up. When an uncleaned unit is compiled, the consistency of the symbol resources is spoiled and the compiler starts parsing resources again. Moreover, it leaves the units in this inconsistent state, so the next build must begin by rebuilding the resources in a consistent manner.

Once the cleanup is complete, your application should build at its optimum rate. If you are already pretty clean in your USES statements, you should be getting near optimal performance.

If you haven't already done so, consider switching to the 3.0/C-style separate interfaces instead of using `OSIntf` or `ToolIntf`. Unless you use almost all the files included by these old-style units, you should use the files as separate units, and get only what you need.

In all cases use `Types.p` instead of `MemTypes.p` and `Packages.p` instead of `PackIntf.p.`

Try not to rely blindly on the auto-inclusion feature of the new interfaces. If you let `Packages` include `Types` in one file and then use `Types` before using `Packages` in the next file, you'll get "symbol table churning": compile-time variables will be different and the symbol table resources will have to be rebuilt each time.

Structure your `Make` file so that the units that come first in your USES clause get compiled before later units and the main program. The symbol table resources for a unit are always rebuilt when the unit is compiled. So if you change a unit and the main program, and your `Make` file builds the main program first, the symbol table resources for the unit will get built when the program is compiled, and again when the unit is compiled.

Use the `-p` option every now and then to see how things are going. Maybe you have compile-time variables that are causing symbol table churning, or maybe the resource fork of a file has become corrupt. Maybe you don't have enough memory to create the symbol table resource (MacApp needs more than a 4 MB partition, and use `-mf` with all tools if possible). Most of the possible inefficiencies in reading or writing symbol table resources can be displayed only by use of the `-p` option

If you use MacApp and switch between versions often (`Debug/noDebug` and so on), you can put the directive `{$K $$Shell(ObjApp)}` before the first unit in your USES clauses. This will create the symbol table resources in files in the same directory as the program's object files. So as you switch from `:.Debug Files:` to `:.Non-Debug Files:` the right set of symbol table resources will already be built.

Precheck the Pascal Syntax

You might precompile the code using Pasmat before the Pascal compiler is used, which could be helpful for quickly finding syntax errors in the code without the penalty of running the full compiler. You might define a command key that performs the operation, as in the following:

```
AddMenu MyMenu 'Pasmat {Active}.§/π' '∂
        (       Pasmat <"{Active}.§" >"{TemporaryFile}" ≥ "{ErrorsFile}" ∂
                && Catenate "{TemporaryFile}" > "{Active}.§" ;∂
                Delete -y "{TemporaryFile}" ∂
        ) || Alert <"{ErrorsFile}"'
```

You might also use the `-c` flag with the Pascal compiler for syntax checking only.

Linker Issues

Limit Symbol Table Generation

In general it takes a lot of time for the linker to build the final .SYM file. Try to avoid building symbol files unless needed (like when stuck with a problem in the source code). In many cases the `-msg full` (or `-Names` flag in MacApp) compiler flag for MacsBug name generation might be OK for a test of where the application crashes.

You might also create a limited set of SYMBOLIC (.SYM) information. Here's an example from MacApp 3.0 of how this could be achieved (in the case of general C++ code, just specify '`-sym on`' in the C++ compiler for those files that you need for debugging). This technique will also save both RAM and disk space.

```
################################################################
# L I B R A R Y   D E P E N D E N C I E S
################################################################
"{ObjApp}{LibName}"ƒƒ ∂
   {LibObjs}
   IF {MacAppLibrary} || {LibName} !~ /MacApp.lib/ # Special trick to keep
                                          MacApp libraries from building
   {MAEcho} {EchoOptions} "Libbing:       {LibName}"
   SET XToolStartTime `DATE -n`
#  {MALib} ∂
#  {LibOptions} ∂
#  {OtherLibOptions} ∂
#  {LibObjs} ∂
#  -o "{ObjApp}{LibName}"
   execute "Skinny Lib"      <+++++++++ new script file

File:  Skinny Lib   ----------------------------

directory "{malibraries}.nodebug names sym nosys7:"
delete -i macapp.lib

lib -mf -w -sym off -o macapp.nosym ∂
   Geometry.cp.o PascalString.cp.o Toolbox.cp.o UAppleEvents.cp.o UAssociation.cp.o ∂
   UBusyCursor.cp.o UClipboardMgr.cp.o UCPlusTool.cp.o UDebug.a.o UDebug.cp.o ∂
   UDeskScrapView.cp.o UEditionDocument.cp.o UErrorMgr.cp.o UFailure.a.o UFloatWindow.cp.o ∂
   UGeometry.cp.o UGrabberTracker.cp.o UKeySelectionBehavior.cp.o UMacAppGlobals.cp.o ∂
   UMacAppUtilities.cp.o UMemory.a.o UMenuMgr.cp.o UMenuView.cp.o UPascalTool.p.o ∂
   UPatch.cp.o USection.cp.o USectionMgr.cp.o UStream.cp.o USynchScroller.cp.o ∂
   UProjFileHandler.cp.o UScroller.cp.o UTabTEView.a.o UTabTEView.cp.o ∂
   UTearOffMenuView.cp.o UTECommands.cp.o UTEView.cp.o UTranscriptView.cp.o ∂
   UDependencies.cp.o UDesignator.cp.o UTabBehaviors.cp.o

#lib -mf -w -sym off,NoLabels,NoLines,NoVars -o macapp.justTypes∂ lib -mf -w -sym off -o
macapp.justTypes∂
   UCommand.cp.o UCommandHandler.cp.o MacAppTypes.cp.o UAdorners.cp.o UBehavior.cp.o ∂
   UDrawingEnvironment.cp.o UEvent.cp.o UFile.cp.o UFileHandler.cp.o UMemory.cp.o ∂
   UIterator.cp.o UPopup.cp.o UViewBehavior.cp.o UViewServer.cp.o

lib -mf -w -sym on -o macapp.lib macapp.nosym macapp.justTypes∂
   UApplication.cp.o UControl.cp.o UDialog.cp.o UDialogBehavior.cp.o UDocument.cp.o ∂
   UEventHandler.cp.o UFailure.cp.o UFileBasedDocument.cp.o UGridView.cp.o ∂
   UList.cp.o UObject.cp.o UPascalObject.a.o UPascalObject.cp.o UPrintHandler.cp.o ∂
   UPrinting.cp.o UView.cp.o UWindow.cp.o

delete macapp.nosym macapp.justTypes
```

Δ .SYM generated = 379 s, no .SYM generated = 276 s, savings 27.2%.

Use Libraries If Possible

The linker will perform much faster if you link together library files (created by the Lib tool) than if you separately compile .o files. Consult the latest MPW documentation which describes various ways of using the Lib tool with projects.

CODE Resources

If you have code resources that do not make any intersegment calls (such as standalone code and `XCMD` style code resources), you can use the `Rez` tool to add these resources directly to the binary instead of using the link tool. This should save some time; how much depends on the actual project.

Global Data

The link tool will build a complete image of the globals to be initialized. If the global area is large, this might take a long time. Try to avoid extensive and unneeded use of global data.

C++ Code Issues

Smaller Files Compile Faster

Split huge source and header files into smaller modules and create dependencies in the Makefile that will trigger compilations only when a particular file has changed. A known caveat with C++ is the `vtable` consistency. Sometimes the vtables have to be created from scratch in order to synchronize the `vtable` information. If the compilation and linking phase has generated a binary, but when you are running the application it has problems, try to recompile most or maybe all sources for a quick test in order to see if the problem has to do with vtables.

Don't Include All Class Headers

If possible place include statements with C++ classes internally used in the .cp file instead of in the header (.h) file. When developers are using a particular header file, they don't need to include class headers that are not needed, and this saves some time. In general try to avoid unnecessary inclusions of classes.

C++ Dump/Load

One problem with compiling object-oriented programs has to do with the parsing of header files. Generally, 80% of the compile time is spent parsing header files. However, most header files remain unchanged for long periods of time during the programming phase. So the header files are reread and reparsed, time after time.

C++ dump/load solves this problem by dumping the header file information to a single file. During compilation of class methods, the compiler loads from this dump file each time it needs the header file information. You can get even more speed by placing all the dump and object files on a RAM disk.

To use dump/load you need to decide which header files are static and not subject to change. For MacApp, the obvious choice is the MacApp class header files. For other complex frameworks, consider only the most stable header files for the dump file. If you alter header files often, the compiler has to create a new dump file, and the dumping process takes a long time.

For more general C++ dump/load guidelines, please consult the MPW C++ Release Notes.

MacApp Code Issues

Using Dump/Load With MacApp

`MABuild` has a flag called `-CPlusLoad`. When this is present, the C++ compiler dumps the MacApp header file information to a folder called `Load Files` inside the MPW folder. This happens during the first compile only. A dump file can take 1 to 2 MB of space, so check your disk space before doing the dump. Also remember that if you have many release versions of the same header files that are dumped, you need to delete the earlier dump files; otherwise you will encounter mysterious bugs.

Dump/load requires lots of heap space for the tools, so now is the time to start using the `-mf` option with `CPlus`, `Link`, and `Lib`. Or increase the MPW application heap size—depending on the size of your sources, up to 4 MB or more. If the CFront tools don't have enough memory for the memory-consuming part of the parsing, error messages such as *"free store exhausted"* will be displayed.

MacApp has a special startup file in the MacApp folder where you can specify default settings. One of the variables defined in this file is `MABuildDefaults`:

```
# SET MABuildDefaults "{MABuildDefaults} -PasLoad -CPlusLoad"
```

Uncomment this line and restart MPW, or select and execute the command. The next time you build your MacApp application, `MABuild` will automatically use the MacApp header files to create a dump file inside the '`MPW:Load Files`' folder. Note that this is now the default case with MacApp 3.0.

You can go one step further and specify that additional header files should be dumped. To do this, edit the `MacApp:Tools` file called `'Build Rules and Dependencies.'` Here's an excerpt from that file:

```
# Load/Dump files must be kept current for C++ too
{CPlusLoadFiles}      ƒ      {MacAppCPlusIntf}
        {MAEcho} {EchoOptions} "C++ Load/Dump: UMacApp.h.dump"
        IF `EXISTS {CPlusLoad}` != "
                Delete {CPlusLoad}
                END
        {MACPlus} ∂
                {CPlusOptions} ∂
                {OtherCPlusOptions} ∂
                -i "{SrcApp}" ∂
                -i "{MACIncludes}" ∂
                "{MACIncludes}UMacApp.h" ∂
                -mf ∂
                # Any other files you want to include in the dump
                        could go here ∂
                -dumpc {CPlusLoad} || (Delete {CPlusLoad})
```

If you're sure that you will repeatedly include certain additional header files in the MacApp dump file, you can add them to this file. Any building block headers (`U≈.h` files) that are likely to be stable are good candidates.

You can also define build rules for dump/load files in the MAMake file for each MacApp project. This way you can have different dump/load definitions for various permutations of source code and header code combinations. Writing your own MAMake dump rules gives you better control over what is to be dumped. Instead of generically dumping all MacApp header files, you can dump only those MacApp and application header files actually used— you don't have to dump all the MacApp header files as MacApp's default C++ dump does. This saves time and disk space.

This works well for handling header files that are part of your own project as long as you don't frequently change the header files, which triggers a costly dump operation. Here are some guidelines:

- You must create a header file dependency rule for the dump file if you want dependencies operating on the header file changes.
- Use `{SrcApp}` prefixes for the application source code file names, and `{ObjApp}` prefixes for the application object code file names.
- You will be overriding most of the basic building rules, so if you want the MPW shell to show what it is doing, add an `Echo` statement as in the original rules.

The C++ Release Notes discuss in great detail how to build the dump file header file. Once again, the trick is to move all static header files to one single file, and call it "`MyAppDump.h`" or something similar. In all the other header files, include the following:

```
#ifndef __MYAPPDUMP__
#include "MyAppDump.h"
#endif __MYAPPDUMP__
```

Do the same `ifndef` trick with the included header files in the dump header file, so that the compiler won't need to include the file many times. Build a rule for dumping the `MyAppDump.h` file, or do it by compiling the header from the MPW command line.

Sometimes a header file is static; then suddenly you are tearing your object framework apart in a frenzy, making incremental changes. A good way to support this would be to use multiple dump files, where sometimes you dump and load from many files, but other times you load from only one dump file, allowing the compiler to parse the header file that is subject to change. (This would be faster when header files are changing, because the dump phase takes a long time.)

Alas, it's not likely that there will ever be support for multiple dump files in MPW C++ because dump files contain structures that are hard to merge. You can achieve a similar functionality using flags inside the header files. You could instruct `MABuild` to dump your own header files in addition to the basic header files. This is done with a programmer-defined MABuild option:

```
MABuild -d qOwnDump=TRUE ...
```

This `qOwnDump` flag controls use of dump files within the header files via a simple `#ifdef qOwnDump` directive (see sample code in the Snippets collection (Developer CD, ETO CD, AppleLink, ftp.apple.com). By using this directive, you can exclude your header files from the dump phase while incrementally changing header files; when again working with method implementations, so that headers are static, you can again dump your header files.

△ No dump files = 818 s, dump file† = 660s, savings 19.3%.

†Taken from earlier MacApp 2.0.1 load/dump testing with DemoText

MPW Issues

More RAM Memory

More memory means more application heap space, and this means less segment loading in cases where segments are purged out of memory in memory-tight situations. If the MPW memory partition is big enough most tools could stay in the MPW heap, and this improves the performance, but not much! Note that you don't need to go overboard with the application heap space. The peak parts of memory use could be handled with the `-mf MultiFinder` temporary memory flag which is implemented with our compilers, linker, and `Lib` tools. For instance a 4Mb MPW partition is suitable for MacApp programming if the `-mf` flag is defined for the compilers linker and Lib.

Δ 4Mb heap = 379 s, 12Mb heap = 379 s, difference 0%.†

† 5 test runs each!

RAM Disks

To avoid file I/O bottlenecks you might think about using a RAM disk. The following order is based on the list of the most important folders/files, and if you have more RAM disk space you could include more from this list until you have most of the development environment and the sources on the RAM disk (the most extreme case). In some cases, like the first three examples, all you need to do is to redefine the exported value in an MPW startup file (the last one!), as in:

```
Set CPlusScratch "RAMDisk:"
Export CPlusScratch
```

In other cases you need to copy the files/folders to the RAM disk, and add the paths to the new folder in such a way that the MPW environment will look into the particular folder first , as in:

```
Set Commands "RAMDisk:Tools:,{Commands}"
Export Commands
```

Here's the recommended list:

`{MATemporaries}`	temporary folder for files that MacApp MaBuild creates
`{CPlusScratch}`	temporary folder for files that MPW C++ creates
`{MALoadFiles}`	MacApp dump/load files folder
`MPW:Tools`	tools (like compilers) for faster load into memory
`MPW:Scripts`	scripts, for faster load into memory
`{Libraries}`	general MacOS libraries, for faster load

`{CLibraries} or`
`{PLibraries}`
	general C/Pascal libraries for faster load

`{MALibraries}`	MacApp libraries (.lib, .rsrc files)

. . . your own project files . . .

Any other possible additions are MPW and MacApp header files and the actual MPW shell itself, including any other development tools. However, these take a lot of space, so we are talking about a huge RAM disk. If you have a +50 MB RAM disk, you might even place the whole MPW and MacApp folders on the RAM disk, which is the quickest way to get the benefits of such a large RAM disk. However, you then need a RAM disk utility which will save and restore the contents if the system is shut down.

Δ No RAM disk = 379 s, 4Mb RAM disk† = 342 s, savings 9.8%.
†{MATemporaries}, {CplusScratch}, {MALoadFiles}

Testing

The following MPW script is useful for testing purposes:

```
Echo -n > "{MPW}Dump"                           # specify output file/window
Open "{MPW}Dump"

for cases in 1 2 3 4 5                          # define how many tests
      Echo "Test Number "{cases}
      set StartTime `Date -n`
      set exit 0
      MaBuild -debug -sym Calc  ΣΣ "{MPW}Dump"  # place whatever job here
      delete -y •Debug≈                         # clean up afterwards
      set exit 1
      set TimeNow `Date -n`
      set Elapsed `Evaluate {TimeNow}-{StartTime}`
      set Elapsed "`Date -c {Elapsed} -t`"
      If "{Elapsed}" =~ /12:([0-9]+:[0-9]+)®1 [AP]M/
       Set Elapsed "0:{®1}"
      Else If "{Elapsed}" =~ /0*([0-9]+:[0-9]+:[0-9]+)®1 [AP]M/
      Set Elapsed "{®1}"
      End
      Echo "∂t◊ Build time: {Elapsed}"
End
```

Conclusion

The four most valuable performance improvements are:

I. RAM disk use (the more you could place on the RAM disk, the better performance)
II. Don't compile and link with the `-sym on` option unless needed
III. Use libraries
IV. Avoid compiling/linking, use tools which will postpone unneeded compilation and linking

Use common sense and consider whether a particular scheme will require more resources and/or memory. Carefully follow Apple announcements about new tools and development environments that might fix bugs that have caused slower performance, or brand-new tools that address performance issues.

Credits

Thanks to the following contributors (listed in a twisted order, where the sort algorithm is an NP-complete problem): Jack Robson, Keith Rollin, Larry Rosenstein, Bryan Stearns, Blue Meanies, Chris Knepper, Rich Norling, Karl Goiser, Pete Richardson, Greg Robbins, Jim Reekes, programmers on MacApp.Tech$, Jeff Sandvik.

Further Reference:

- MPW Documentation

NuBus™ is a trademark of Texas Instruments.

#314: OmegaSANE

Written by: Dave Radcliffe and Colin McMaster May 1992

System 7.0.1 introduced a new version of SANE (the Standard Apple Numerics Environment) known as OmegaSANE. This Note discusses the features of OmegaSANE and the associated compatibility risks. This note covers:

- OmegaSANE features, including:
  - Correctly rounded binary ↔ decimal conversions
  - Faster transcendental functions
  - Backpatching of Pack 4 SANE traps for faster package entry
- Compatibility risks due to backpatching

Introduction

System 7.0.1 introduced a new version of the Standard Apple Numerics Environment (SANE) package referred to as OmegaSANE (ΩSANE). While it improves the performance of SANE, it is not without compatibility risks, which are detailed below. New binary ↔ decimal conversions have been included that are correctly rounded across the entire range of double extended. This will result in incompatibility with previous conversion algorithms for a variety of input values; the results of the ΩSANE conversions are uniformly more accurate, and will be compatible with future releases of SANE.

ΩSANE Features

The ΩSANE release provides a number of performance enhancements while maintaining conformance with SANE. The following enhancements have been implemented:

- Correctly rounded binary ↔ decimal conversions
- Faster transcendental functions
- Backpatching of SANE traps for faster package entry

Tables showing the performance of ΩSANE are given below. A key aspect of the performance gain is the "backpatching" of SANE traps. This mechanism will be discussed below under "Pitfalls."

The version of ΩSANE supplied with System 7.0.1 installs only on machines equipped with a Macintosh IIci ROM or later and also equipped with an FPU. For example, it installs on a Macintosh IIsi running System 7.0.1 and equipped with an FPU, but not on one without an FPU. On the Macintosh Quadras and the PowerBook 140/170 it is in ROM, although it doesn't load on the PowerBook 140 unless it has an optional FPU. On machines where the FPU is optional, addition of an FPU may require re-installation of System 7.0.1 before the FPU version of ΩSANE will load.

Table 1 presents configuration information concerning the versions of SANE which may be installed by System 7.0.1. Information in this table is subject to change. The FPU column states whether the machine has an FPU or if it is optional. The column labeled "Correctly Rounded Bin ↔ Dec" shows whether improved versions of the binary ↔ decimal conversion routines (discussed below) are available, and which version (V.1 or V.2) is supplied. The "ΩSANE FPU Version" column states whether ΩSANE will load on a given configuration.

Note: The information in this table will undoubtedly change in the future. Developers should not assume that any particular machine/system software combination does or does not have ΩSANE.

Table 1 SANE Configurations

Macintosh CPU	FPU	Correctly Rounded Bin↔Dec	ΩSANE FPU Version
Mac Plus	No	No	No
Mac SE	No	No	No
Mac Classic	No	No	No
Mac Classic II	Optional	V.2 in ROM	w/ 7.0.1 & FPU†
Mac LC	Optional	V.1 in ROM	w/ 7.0.1 & FPU†
Mac IIsi	Optional	V.1 in ROM	w/ 7.0.1 & FPU†
Mac SE/30	Yes	No	No
Mac II	Yes	No	No
Mac IIx	Yes	No	No
Mac IIcx	Yes	No	No
Mac IIci	Yes	V.2 with 7.0.1	w/ 7.0.1
Mac IIfx	Yes	V.2 with 7.0.1	w/ 7.0.1
PowerBook 100	Yes	No	No
PowerBook 140	Optional	V.2 in ROM	w/ 7.0.1 & FPU†
PowerBook 170	Yes	V.2 in ROM	Yes
Quadra 700	Yes	V.2 in ROM	Yes
Quadra 900	Yes	V.2 in ROM	Yes

† FPU optional systems may require reinstallation of System 7.0.1 after installation of an FPU before the ΩSANE FPU version will load.

There is no way programmatically to disable ΩSANE in System 7.0.1, nor is it a user option. There is also no way to reliably detect that ΩSANE is present.

Binary ↔ Decimal Conversions

ΩSANE includes binary ↔ decimal conversion routines that may produce results that differ from previous versions of SANE. Versions of these routines have been in some machine ROMs since the Macintosh IIsi, and an improved version (V.2) is in the newest ROMs, as well as ΩSANE. Refer to Table 1 for current configuration information. As a result of these improved conversion routines, floating-point constants that are used in the body of source code will compile differently in the presence of ΩSANE than with older SANE engines. The results are uniformly better, but may cause unexpected variances from test suites (for instance).

Therefore, care must be given to the arithmetic environment in which compilations are made. One can tell which version of the binary↔decimal conversions is currently in use by performing the following computation on the Calculator DA: 45/100 – 0.45. On older versions of SANE, the result is –2.71051E-20. With ΩSANE, the result is 0 as expected.

Performance Improvements

ΩSANE can significantly improve the performance of many floating-point SANE operations. It does this by replacing _FP68K trap calls with JSR instructions directly into the appropriate SANE code. This is discussed in more detail under "Pitfalls." 7.0.1 ΩSANE does not alter _Elems68K trap calls, but internal code improvements are used to increase the performance of transcendental functions. Finally, ΩSANE does not affect the performance of code compiled to use the FPU directly, although some library routines used with such code (such as the CSANELib881.o routines NextAfter, Classify, Scalb, and Remainder) will be backpatched by ΩSANE because they call _FP68K.

Table 2 shows typical performance improvement using ΩSANE on a Macintosh IIci. Of course, actual performance depends on how heavily you use SANE and the mixture of SANE operations you use.

Table 2 SANE Performance

Operation	Macintosh IIci SANE (in flops)	Macintosh IIci ΩSANE (in flops)	Speedup Factor
Add/Subtract	18,794	59,259	3.15
Multiply	18,182	53,571	2.95
Divide	18,476	55,300	2.99
Square Root (Exact)	18,547	65,934	3.55
Square Root (Inexact)	18,349	59,113	3.22
Cosine	902	7,058	7.82
Sine	1,290	8,108	6.29
Tangent	826	7,185	8.70
Logarithm	820	7,643	9.32
Exponential	723	7,317	10.12
Compound	413	3,324	8.05
Annuity	358	3,191	8.91

Pitfalls

ΩSANE achieves most of its performance improvement through use of a technique known as "backpatching." Use of backpatching introduces a number of compatibility implications. To implement backpatching, the front end of toolbox SANE has been altered to have multiple entry points corresponding to the most important operations, and *your* RAM-based application code is modified *on the fly* replacing traps with speedier `JSRs` to the appropriate entry point. Subsequent execution causes the code to bypass the trap dispatcher and call SANE directly. For example, an archetypal SANE package call looks like this:

```
        pea         fooSrc                ; Push address of the source operand
        pea         barDst                ; Push the address of the destination operand
        move.w      #OpCode,-(sp)         ; Push a SANE opcode word
        _FP68K                            ; Trap to SANE
```

Notice that the last two instructions occupy three words, just enough for the desired JSR. ΩSANE modifies the RAM-image of the application resulting in code like the following:

```
        pea         fooSrc                   ; Push address of the source operand
        pea         barDst                   ; Push the address of the dest. operand
        jsr         'SANE entry point'       ; JSR to appropriate SANE entry point
```

Herein lies a potential problem. There is nothing to guarantee that the instruction immediately prior to the _FP68K trap was actually executed. A program control operation, such as a BRA, could have caused control to be passed to the _FP68K operation without executing the preceding move.w. For instance, an execution sequence such as the following:

```
        pea         subSrc                   ; Push source for subtract
        pea         subDst                   ; Push destination for subtract
        move        #$0002,-(sp)             ; Push subtract extended opcode word
        bra         @1                       ; Branch to _FP68K trap
        .
        .
        .
        pea         addSrc2                  ; Push source for add
        pea         addDst2                  ; Push destination for add
        move        #$0000,-(sp)             ; Push add extended opcode word
@1      _FP68K                               ; Trap to SANE
```

would have the unseemly result of "backpatching" a JSR to the extended add entry point, and future executions of the subtraction code would branch to the middle of a JSR causing an illegal instruction error (the illegal instruction error might not occur right away). Note, however, that the following similar code sequence works correctly in the presence of backpatching:

```
        pea         addSrc                   ; Push source for add
        pea         addDst                   ; Push destination for add
        bra         @1                       ; Branch to _FP68K trap
        .
        .
        .
        pea         addSrc2                  ; Push source for add
        pea         addDst2                  ; Push destination for add
@1      move        #$0000,-(sp)             ; Push add extended opcode word
        _FP68K                               ; Trap to SANE
```

The type of code sequence that is problematic for backpatching (as above) cannot be emitted by the current MPW C and Pascal compilers or by current Think™ compilers. It also cannot occur with code generated using the macros contained in SANEMacs.a.

Warning -- Although the current generation of MPW compilers create ΩSANE-safe code, the earlier, MPW C 2.0.2 compiler may, under limited circumstances, generate code that has problems with ΩSANE. This is discussed in more detail below.

Additionally, if you have hand coded assembly or code which has been otherwise optimized to use common _FP68K trap instructions you may be at risk from backpatching. We recommend you adjust your code to conform to the prototypical sequence above as there is no performance penalty involved.

Another common programming practice is illustrated in the following example:

```
                 move.w     #$0000,D0            ; Move extended add opcode word to D0
                 bra        @1                   ; Branch to shared backend for SANE trap call
                 .
                 .
                 .
                 move.w     #$0002,D0            ; Move extended subtract opcode word to D0
                 bra        @1                   ; Branch to shared backend for SANE trap call
                 .
                 .
                 .
        @1       move.w     D0,-(SP)             ; Push SANE opcode word on stack
                 _FP68K                          ; Trap to SANE
```

This code operates correctly in the presence of ΩSANE, but is clearly not backpatchable. If you have code that does this, you might consider rewriting it to obtain the performance increase, but the code will work as is.

MPW C 2.0.2

The MPW C 2.0.2 compiler may, under limited circumstances, generate code that works incorrectly under ΩSANE. Apple recommends that all developers use the latest development tools, but as conversion of source may be difficult and time consuming in some cases, below is a description and workaround for the problem. Again, this only affects code compiled without the -mc68881 option.

The following code demonstrates the problem

```
struct foo {
      double data;
      float num;
};

foobar(f,b)
struct foo *f;
unsigned char b;
{
      extended num;

      num = 3.14159;
      if (b)
            f->data = num;          /* FP operation ends block */
      else
            f->num = num;           /* Another FP operation ends block */

      return;
}
```

The critical combination of events occurs when more than one block in an if-else or switch statements ends with a floating-point operation or conversion. The compiler tries to optimize the final floating-point operation by pushing a selector on the stack then branching to a common `_FP68K` trap. This is a classic example of code, like that cited above, that works incorrectly under ΩSANE.

The workaround is to eliminate the final floating-point operation. Here is one way to do this:

```
foobar(f,b)
struct *f;
unsigned char b;
{
        extended num;
        int idummy;

        num = 3.14159;
        if (b) {
                f->data = num;
                idummy = 1;         /* End if block with a non-FP operation */
        } else {
                f->num = num;
                idummy = 0;         /* End else block with a non-FP operation */
        }

        return;
}
```

In this case, each floating-point operation gets its own `_FP68K` trap so there are no problems.

Other Pitfalls

The backpatching performed by ΩSANE can have other implications for developers. For example, applications that checksum their code segments (perhaps to check for viruses) will detect that the segment has been modified. Such applications should checksum segments only at application startup or when the segments are loaded, not after they have been executed.

Likewise, an application must not create a situation which will cause the Resource Manager to write a code segment out to disk after it has been executed. If such a segment been modified by ΩSANE, it can fail when subsequently loaded into memory. Since `CloseResFile` is called when the application quits it will automatically write out changed resources (i.e. if the `resChanged` attribute is true) when closing the file.

Further Reference:

- *Apple Numerics Manual,* Second Edition

Macintosh
Technical Notes

Developer Technical Support

#315: Resolving Alias Files Quietly

Written by: Greg Robbins May 1992

ResolveAliasFile always presents the user identity dialog when mounting remote volumes. This Technical Note offers an alternative function, ResolveAliasFileMountOption, which uses the previously undocumented FollowFinderAlias trap to resolve alias files only if their target is on an already mounted volume. Also included is an IsAliasFile routine for identifying alias files.

Introduction

Finder alias files are one aspect of the Macintosh human interface considered "reserved for users." The internal format of Finder alias files is intentionally undefined because it is subject to change and because Finder alias files should be neither created nor altered by applications. The Finder is the user's domain, and Finder alias files are a user convenience.

Most applications do not need to take special steps to accommodate Finder alias files. They are resolved by the Finder before they are passed to an application in an Open Documents Apple event, as well as by the Standard File Package when it creates the reply record. Occasionally an application may need to resolve alias files manually. That is normally done by calling ResolveAliasFile, as documented in Chapter 9 of *Inside Macintosh* Volume VI.

If a Finder alias file resolves to an item on an unmounted remote volume, ResolveAliasFile will attempt to mount the volume to resolve the alias. For servers, this will bring up the user identity dialog box shown on page 7-24 of *Inside Macintosh* Volume VI. For removable volumes, this will raise the "Please insert…" dialog. Presentation of a dialog may be inappropriate for the application. For example, a Standard File hook procedure that quietly opens the selected file in order to offer a preview of its contents would not want the dialog presented whenever the user selects an alias to a remote item.

Keeping Quiet

The ResolveAliasFileMountOption function listed below allows an application to resolve a Finder alias file only if the alias file's target is on a currently mounted volume. If the target is unavailable and the mountRemoteVols parameter is false, ResolveAliasFileMountOption returns the error nsvErr. ResolveAliasFileMountOption operates like ResolveAliasFile if mountRemoteVols is true. ResolveAliasFileMountOption updates the alias file if necessary, and returns fnfErr if the alias file is part of a circular chain.

ResolveAliasFileMountOption uses the previously undocumented trap FollowFinderAlias to resolve the alias file's 'alis' resource. This is preferable to passing the 'alis' to MatchAlias because it makes no assumptions about how the alias was created or how it should be resolved. Finder aliases may actually be relative aliases rather than direct aliases. In any case, FollowFinderAlias will take the steps necessary to resolve them properly.

FollowFinderAlias should not be used except when necessary to resolve the 'alis' resource of a Finder alias file. Aliases created by applications should be resolved with the Alias Manager calls ResolveAlias and MatchAlias.

Note: FollowFinderAlias is used internally by Apple Computer, Inc. It has not been tested for use by application software. While we do not anticipate any problems, it is the responsibility of the developer to ensure that it operates appropriately and reliably for their application.

ResolveAliasFileMountOption uses the function IsAliasFile, also listed below, to determine if a file is an alias file. In keeping with the interface of ResolveAliasFile, IsAliasFile will indicate if the specified item is a folder rather than a file.

Quiet Calls

To determine if an FSSpec points to an alias file, a folder, or neither, use IsAliasFile.

```
FUNCTION IsAliasFile(fileFSSpec: FSSpec;
                     VAR aliasFileFlag: BOOLEAN;
                     VAR folderFlag: BOOLEAN): OSErr;
```

IsAliasFile simply calls PBGetCatInfo to check if the FSSpec's target has its directory or isAlias bits set. These are described in *Inside Macintosh* Volume IV, page 122, and *Inside Macintosh* Volume VI, page 9-36.

To resolve an alias file without any dialogs appearing, use ResolveAliasFileMountOption.

```
FUNCTION ResolveAliasFileMountOption(VAR fileFSSpec: FSSpec;
                                     resolveAliasChains: BOOLEAN;
                                     VAR targetIsFolder: BOOLEAN;
                                     VAR wasAliased: BOOLEAN;
                                     mountRemoteVols: BOOLEAN): OSErr;
```

The first four parameters of ResolveAliasFileMountOption are the same as those of ResolveAliasFile. fileFSSpec is the specification for a file, alias file, or folder. resolveAliasChains should be true if the resolution should follow down a chain of alias files. targetIsFolder is a return parameter that is set if the fileFSSpec points to or resolves to a folder. wasAliased returns true if the input fileFSSpec was for an alias file.

If the mountRemoteVols parameter is true, ResolveAliasFileMountOption will attempt to mount a volume if necessary to resolve an alias file, making the call equivalent to ResolveAliasFile. If mountRemoteVols is false and the file spec is for an alias file that resolves to a volume not currently mounted, the call will return nsvErr rather than attempt to mount it.

The FollowFinderAlias trap is intended only for resolving alias records obtained from Finder alias files.

```
FUNCTION FollowFinderAlias(fromFile: FSSpecPtr;
                           alias: AliasHandle;
                           logon: BOOLEAN;
                           VAR target: FSSpec;
                           VAR wasChanged: BOOLEAN): OSErr;

    INLINE $700F,$A823;  { MOVEQ #$0F,D0; _AliasDispatch; }
```

fromFile is a pointer to a file for a first attempt at a relative resolution; pass a pointer to the alias file's FSSpec for this. alias is a handle to the alias record taken from the alias file's resources. If logon is true, the alias manager will attempt to mount a volume if necessary to complete the resolution. target will be the FSSpec found by the resolution. If wasChanged is true following the call, FollowFinderAlias has updated the alias record, and the caller should call ChangedResource and WriteResource if the updated record is to be saved in the resource file.

FollowFinderAlias does a single resolution; it does not follow a chain of alias files. FollowFinderAlias returns the same errors as MatchAlias.

MPW Pascal

```
{*-------------------*
 | IsAliasFile       |
 *-------------------*}

FUNCTION IsAliasFile(fileFSSpec: FSSpec;
                     VAR aliasFileFlag: BOOLEAN;
                     VAR folderFlag: BOOLEAN): OSErr;
{ sets aliasFileFlag if the FSSpec points to an alias file;
  sets folderFlag if the FSSpec points to a folder }

  CONST
    kAliasFileBit = 15; { bit of FInfo.fdFlags indicating alias file }
    kDirBit = 4;        { bit of CInfoPBRec.ioFlAttrib indicating directory }

  VAR
    myCInfoPBRec: CInfoPBRec;
    retCode: OSErr;

  BEGIN
    { if called from C we could accidentally be passed nil parameters }
    IF (@fileFSSpec = NIL) OR (@aliasFileFlag = NIL) OR (@folderFlag = NIL) THEN
      BEGIN
        IsAliasFile := paramErr;
        Exit(IsAliasFile);
      END;

    aliasFileFlag := FALSE;
    folderFlag := FALSE;

    { get the item's catalog information }
    WITH myCInfoPBRec DO
      BEGIN
        ioCompletion := NIL;
        ioNamePtr := @fileFSSpec.name;
        ioVRefNum := fileFSSpec.vRefNum;
        ioFDirIndex := 0;
        ioDirID := fileFSSpec.parID;
        ioFVersNum := 0;  { MFS compatibility; see Tech Note #204 }
      END;
    retCode := PBGetCatInfoSync(@myCInfoPBRec);

    IF retCode = noErr THEN

      { set aliasFileFlag if the item is not a directory and the
        aliasFile bit is set }
      BEGIN
        IF BTst(myCInfoPBRec.ioFlAttrib, kDirBit) THEN
          folderFlag := TRUE
        ELSE IF BTst(myCInfoPBRec.ioFlFndrInfo.fdFlags, kAliasFileBit) THEN
          aliasFileFlag := TRUE;
      END;

    IsAliasFile := retCode;
  END;


{*-----------------------------*
 | ResolveAliasFileMountOption |
 *-----------------------------*}

FUNCTION ResolveAliasFileMountOption(VAR fileFSSpec: FSSpec;
                                     resolveAliasChains: BOOLEAN;
                                     VAR targetIsFolder: BOOLEAN;
                                     VAR wasAliased: BOOLEAN;
                                     mountRemoteVols: BOOLEAN): OSErr;

{ ResolveAliasFileMountOption operates identically to ResolveAliasFile,
  except that if mountRemoteVols is false, no attempt will be made to
  resolve aliases that point to items on non-local volumes }

{ if mountRemoteVols is false, ResolveAliasFileMountOption returns nsvErr if
  fileFSSpec points to an unmounted volume }

{ this routine requires the Alias Manager, available under System 7 }

  CONST
    kAliasFileBit = 15;  { bit of FInfo.fdFlags indicating alias file }
    kMaxChains = 10;  { maximum number of aliases to resolve before giving up }

  VAR
    myResRefNum, chainCount: INTEGER;
    alisHandle: Handle;
    initFSSpec: FSSpec;
    updateFlag, foundFlag, wasAliasedTemp, specChangedFlag: BOOLEAN;
    retCode: OSErr;

  FUNCTION FollowFinderAlias(fromFile: FSSpecPtr;
                             alias: AliasHandle;
                             logon: BOOLEAN;
                             VAR target: FSSpec;
                             VAR wasChanged: BOOLEAN): OSErr;

    INLINE $700F,$A823;  { MOVEQ #$0F,D0; _AliasDispatch; }

  { FollowFinderAlias resolves an alias taken from a Finder alias file,
    updating the alias record (but not the alias resource in the file) if
    necessary.

    Warning: This trap is used internally by Apple Computer, Inc.
      It has not been tested for use by application software.
      While we do not anticipate any problems, it is the responsibility
      of the developer to ensure that it operates appropriately and
    reliably for their application.

  fromFile is a pointer to a file for a first attempt at a relative search
  (pass the alias file's FSSpec); alias is a handle for the alias record
  taken from the file's resources; the alias manager will attempt to mount
  a volume if logon is TRUE; target is the found FSSpec; wasChanged is set
  to TRUE if the alias record needs updating.

  FollowFinderAlias does a single resolution; it does not follow a chain of
  alias files.

  FollowFinderAlias returns the same errors as MatchAlias. }


BEGIN { ResolveAliasFileMountOption }

  { check parameters }
  IF (@fileFSSpec = NIL) OR (@targetIsFolder = NIL) OR (@wasAliased = NIL) THEN
    BEGIN
      ResolveAliasFileMountOption := paramErr;
      Exit(ResolveAliasFileMountOption);
    END;

  initFSSpec := fileFSSpec; { so FSSpec can be restored in case of error }
  chainCount := kMaxChains; { circular alias chain protection }
  targetIsFolder := FALSE;
  foundFlag := FALSE;
  specChangedFlag := FALSE; { in case of error, restore file spec if it changed }
  myResRefNum := -1;        { resource file not open }

  { loop through chain of alias files }
  REPEAT

    chainCount := chainCount - 1;

    { check if FSSpec is an alias file or a directory }
    { note that targetIsFolder => NOT wasAliased }
    retCode := IsAliasFile(fileFSSpec, wasAliased, targetIsFolder);
    IF (retCode <> noErr) OR (NOT wasAliased) THEN Leave; { break from loop }

    { get the resource file reference number }
    myResRefNum := FSpOpenResFile(fileFSSpec, fsCurPerm);
    retCode := ResError;
    IF myResRefNum = -1 THEN Leave;

    { the first 'alis' resource in the file is the appropriate alias }
    alisHandle := Get1IndResource(rAliasType, 1);
    retCode := ResError;
    IF alisHandle = NIL THEN Leave;

    { load the resource explicitly in case SetResLoad(FALSE) }
    LoadResource(alisHandle);
    retCode := ResError;
    IF retCode <> noErr THEN Leave;

    retCode := FollowFinderAlias(@fileFSSpec, AliasHandle(alisHandle),
      mountRemoteVols, fileFSSpec, updateFlag);
    { FollowFinderAlias returns nsvErr if volume not mounted }

    IF retCode = noErr THEN
      BEGIN
        IF updateFlag THEN
          { the resource in the alias file needs updating }
          BEGIN
            { we don't care if these cause errors, which they may
              do if we don't have write permission }
              ChangedResource(alisHandle);
              WriteResource(alisHandle);
            END;

          specChangedFlag := TRUE; { in case of error, restore file spec }

          retCode := IsAliasFile(fileFSSpec, wasAliasedTemp, targetIsFolder);

          IF retCode = noErr THEN
            { we're done unless it was an alias file and we're
              following a chain }
            foundFlag := NOT (wasAliasedTemp AND resolveAliasChains);
        END;

      CloseResFile(myResRefNum);
      myResRefNum := -1;

    UNTIL (retCode <> noErr) OR (chainCount = 0) OR (foundFlag);

    { return file not found error for circular alias chains }
    IF (chainCount = 0) AND (NOT foundFlag) THEN retCode := fnfErr;

    { if error occurred, close resource file and restore the original FSSpec }

    IF myResRefNum <> -1 THEN CloseResFile(myResRefNum);
    IF (retCode <> noErr) AND (specChangedFlag) THEN fileFSSpec := initFSSpec;

    ResolveAliasFileMountOption := retCode;
  END;
```

MPW C

```
/*-------------*
 | IsAliasFile |
 *-------------*/

pascal OSErr IsAliasFile(const FSSpec *fileFSSpec,
                         Boolean *aliasFileFlag,
                         Boolean *folderFlag)
/* sets aliasFileFlag if the FSSpec points to an alias file;
   sets folderFlag if the FSSpec points to a folder */

{
  CInfoPBRec myCInfoPBRec;
  OSErr retCode;

  if (fileFSSpec == nil || aliasFileFlag == nil || folderFlag == nil)
    return paramErr;

  *aliasFileFlag = *folderFlag = false;

  /* get the item's catalog information */
  myCInfoPBRec.hFileInfo.ioCompletion = nil;
  myCInfoPBRec.hFileInfo.ioNamePtr = &fileFSSpec->name;
  myCInfoPBRec.hFileInfo.ioVRefNum = fileFSSpec->vRefNum;
  myCInfoPBRec.hFileInfo.ioDirID = fileFSSpec->parID;
  myCInfoPBRec.hFileInfo.ioFVersNum = 0;  /* MFS compatibility, see TN #204 */
  myCInfoPBRec.hFileInfo.ioFDirIndex = 0;

  retCode = PBGetCatInfoSync(&myCInfoPBRec);

  /* set aliasFileFlag if the item is not a directory and the
     aliasFile bit is set */
  if (retCode == noErr) {
    /* check directory bit */
    if ((myCInfoPBRec.hFileInfo.ioFlAttrib & ioDirMask) != 0)
      *folderFlag = true;

    /* check isAlias bit */
    else if ((myCInfoPBRec.hFileInfo.ioFlFndrInfo.fdFlags & 0x8000) != 0)
      *aliasFileFlag = true;
  }

  return retCode;
}


/*--------------------*
 | FollowFinderAlias |
 *--------------------*/

pascal OSErr FollowFinderAlias(const FSSpec *fromFile,
                               AliasHandle alias,
                               Boolean logon,
                               FSSpec *target,
                               Boolean *wasChanged)

  = {0x700F, 0xA823};  /* MOVEQ #$0F,D0; _AliasDispatch; */

/*  FollowFinderAlias resolves an alias taken from a Finder alias file,
    updating the alias record (but not the alias resource in the file) if
    necessary.

    Warning: This trap is used internally by Apple Computer, Inc.
      It has not been tested for use by application software.
      While we do not anticipate any problems, it is the responsibility
      of the developer to ensure that it operates appropriately and
      reliably for their application. */

/*-----------------------------*
 | ResolveAliasFileMountOption
 *-----------------------------*/

pascal OSErr ResolveAliasFileMountOption(FSSpec *fileFSSpec,
                                         Boolean resolveAliasChains,
                                         Boolean *targetIsFolder,
                                         Boolean *wasAliased,
                                         Boolean mountRemoteVols)
{
/* maximum number of aliases to resolve before giving up */
#define MAXCHAINS 10

  short myResRefNum;
  Handle alisHandle;
  FSSpec initFSSpec;
  Boolean updateFlag, foundFlag, wasAliasedTemp, specChangedFlag;
  short chainCount;
  OSErr retCode;

  if (fileFSSpec == nil || targetIsFolder == nil || wasAliased == nil)
    return paramErr;

  initFSSpec = *fileFSSpec; /* so FSSpec can be restored in case of error */
  chainCount = MAXCHAINS;   /* circular alias chain protection */
  myResRefNum = -1;         /* resource file not open */

  *targetIsFolder = foundFlag = specChangedFlag = false;
  /* loop through chain of alias files */
  do {
    chainCount--;

    /* check if FSSpec is an alias file or a directory */
    /* note that targetIsFolder => NOT wasAliased      */

    retCode = IsAliasFile(fileFSSpec, wasAliased, targetIsFolder);
    if (retCode != noErr || !(*wasAliased)) break;

    /* get the resource file reference number */
    myResRefNum = FSpOpenResFile(fileFSSpec, fsCurPerm);
    retCode = ResError();
    if (myResRefNum == -1) break;

    /* the first 'alis' resource in the file is the appropriate alias */
    alisHandle = Get1IndResource(rAliasType, 1);
    retCode = ResError();
    if (alisHandle == nil) break;

    /* load the resource explicitly in case SetResLoad(FALSE) */
    LoadResource(alisHandle);
    retCode = ResError();
    if (retCode != noErr) break;

    retCode = FollowFinderAlias(fileFSSpec, (AliasHandle) alisHandle,
      mountRemoteVols, fileFSSpec, &updateFlag);
    /* FollowFinderAlias returns nsvErr if volume not mounted */

    if (retCode == noErr) {

      if (updateFlag) {
        /* the resource in the alias file needs updating */
        ChangedResource(alisHandle);
        WriteResource(alisHandle);
      }

      specChangedFlag = true; /* in case of error, restore file spec */

      retCode = IsAliasFile(fileFSSpec, &wasAliasedTemp, targetIsFolder);
      if (retCode == noErr)
        /* we're done unless it was an alias file and we're following a chain */
        foundFlag = !(wasAliasedTemp && resolveAliasChains);

    }

    CloseResFile(myResRefNum);
    myResRefNum = -1;

  } while (retCode == noErr && chainCount > 0 && !foundFlag);

  /* return file not found error for circular alias chains */
  if (chainCount == 0 && !foundFlag) retCode = fnfErr;

  /* if error occurred, close resource file and restore the original FSSpec */

  if (myResRefNum != -1) CloseResFile(myResRefNum);
  if (retCode != noErr && specChangedFlag) *fileFSSpec = initFSSpec;

  return retCode;
}
```

Further Reference:

- *Inside Macintosh*, Volume VI, Alias Manager
- *Inside Macintosh*, Volume VI, Finder Interface, pp. 9-29 to 9-32

#316: Data Access Extensions

Written by: Chuq Von Rospach and Dan Strnad May 1992

This Technical Note discusses coding data access extensions that provide an interface between the Data Access Manager and remote data sources. Each of the functions that a data access extension must implement is described.

Introduction

A data access extension is a program that provides an interface between the Data Access Manager and the remote data source. The data access extension implements all of the low-level functions and handles network communication for the Data Access Manager. Because the data access extension implements the low-level Data Access Manager functions, it must return appropriate result codes and handle asynchronous execution of functions as appropriate.

Note: Each data access extension contains a flag that indicates to the Data Access Manager whether the data access extension supports asynchronous execution of routines. If an application attempts to make an asynchronous call to a data access extension that has the first bit (bit 0) of the flags field cleared to 0, the Data Access Manager returns a result code of `rcDBAsyncNotSupp` and terminates execution of the routine. To ensure compatibility of your data access extension with all applications, your data access extension should support asynchronous execution of functions. The data access extension flags field is described in the next section, "Contents of a Data Access Extension."

As soon as the data access extension begins execution of an asynchronous routine, it should return a `noErr` result code for the function result and set the result field of the asynchronous parameter block to 1. It should return control to the calling routine as quickly as possible. When it terminates execution of the routine, the data access extension must place the return code in the result field. Result codes for each of the data access extension routines are listed in the next section. The asynchronous parameter block is described in "Asynchronous Execution of Routines" in the Data Access Manager chapter of *Inside Macintosh* Volume VI, page 8-50.

When the data access extension has completed execution of an asynchronous routine, it must call the application's completion routine pointed to by the completionProc field of the asynchronous parameter block. The completion routine is described in "Asynchronous Execution of Routines" in the Data Access Manager chapter of *Inside Macintosh* Volume VI, page 8-50.

The data access extension can use the `ddevRef` field in the asynchronous parameter block for its own purposes.

This Tech Note describes each of the functions that a data access extension must implement.

Contents of a Data Access Extension

A data access extension consists of a file of type 'ddev', located in the Extensions Folder. The data access extension file must contain these resources:

'ddev' (function; resource ID is 128)

'STR ' (name of data access extension; resource ID is 128)

'dflg' (version number and flags; resource ID is 128)

The 'ddev' resource contains a function that implements all of the low-level Data Access Manager functions. The Data Access Manager calls the ddev function whenever the manager needs to execute a low-level function.

Here is a function declaration for a ddev function.

```
FUNCTION MyDDev(params: DDEVParams) : OSErr;
```

The params parameter is a parameter block that includes a routine selector. The data access extension parameter block is described in the next section, "Data Access Extension Parameters."

You must set bit 6 of the resource attribute byte to 1 for the 'ddev' resource so that the resource is read into the system heap. Resource attributes are discussed in the Resource Manager chapter of of *Inside Macintosh* Volume I.

The 'STR ' resource must contain a character string of not more than 63 characters that specifies the name of the data access extension. Under tsystem 7.0 or later, the data access extension is assumed to reside in the "Extensions" folder within the System folder. The data access extension name is a parameter to the DBInit function.

The 'dflg' resource contains two 4-byte fields, as follows:

```
TYPE DDEVFlags =
      RECORD
          version:              LongInt;          {data access extension format}
          flags:                LongInt           {data access extension flags}
      END;
```

The version field indicates the version of the data access extension format used for this data access extension. It must be set to 0 for the version 7.0 Data Access Manager.

The flags field specifies flags that the data access extension must set. At present, only the least significant bit is defined; all other bits must be cleared to 0. Set the flags field to the constant kAsyncSupported (that is, set the least significant bit to 1) if this data access extension supports asynchronous calls, or to 0 if it does not. If an application attempts to make an asynchronous call to a data access extension that has the flags field cleared to 0, the Data Access Manager returns a result code of rcDBAsyncNotSupp.

Data Access Extension Parameters

This section describes the parameter block that the Data Access Manager passes to a data access extension. The section "Data Access Extension Messages" specifies which parameters are significant for each type of routine and whether each value is passed to the data access extension or returned by the data access extension.

The Data Access Manager passes a parameter block to a data access extension. The parameter block is defined as a `DDEVParams` record.

```
TYPE DDEVParams =
      RECORD
          message:               Integer;          {routine selector}
          ddevStorage:           LongInt;          {storage for use by }
                                                   { data access extension}
          asyncPB:               DBAsyncParmBlkPtr;
                                                   {pointer to asynch }
                                                   { parameter block}
          sessID:                LongInt;          {session ID}
          returnedID:            LongInt;          {session ID returned}
          version:               LongInt;          {version number}
          start:                 LongInt;          {session start time}
          host:                  StringPtr;        {name of remote system}
          user:                  StringPtr;        {user name}
          password:              StringPtr;        {user password}
          connStr:               StringPtr;        {connection string}
          network:               StringPtr;        {name of the network}
          buffer:                Ptr;              {data buffer}
          err1:                  LongInt;          {primary error code returned}
          err2:                  LongInt;          {secondary error code }
                                                   { returned}
          item1:                 StringPtr;        {pointer to object of error }
                                                   { message}
          item2:                 StringPtr;        {pointer to object of }
                                                   { error message}
          errorMsg:              StringPtr;        {pointer to error message}
          timeout:               LongInt;          {timeout value for DBGetItem}
          dataType:              DBType;           {data type}
          sessNum:               Integer;          {session number}
          state:                 Integer;          {status of the data source}
          len:                   Integer;          {length of data item}
          places:                Integer;          {decimal places in data item}
          flags:                 Integer;          {flags}
          abort:                 Boolean           {flag for DBBreak}
      END;
```

Field Descriptions

`message` The routine selector that tells the data access extension which function to execute. For the values for this field and descriptions of the routines, see the next section, "Data Access Extension Messages."

`ddevStorage` Reserved for use by the data access extension. The Data Access Manager sets this field to 0 when it calls the data access extension with the DBOpen message. The data access extension can store any value in this field at that time, and the Data Access Manager retains that value on all subsequent calls to the data access extension. The value of this field does not depend on the session ID; it is the same for all sessions that are using the same data access extension.

`asyncPB` Pointer to the asynchronous parameter block. If the application is making a synchronous call, this field is NIL. The asynchronous parameter block is described in "Asynchronous Execution of Routines" in the Data Access Manager chapter of *Inside Macintosh* Volume VI.

`sessID` The session ID. The data access extension returns the session ID to the DBInit function; all other Data Access Manager functions pass the session ID to the data access extension.

The purpose of the session ID is to provide applications with a unique identifier for each active session. The Data Access Manager reads the session ID returned by the data access extension, and then assigns a unique session ID to each session. The Data Access Manager performs the mapping between the session IDs that it provides to applications and the ones used by each data access extension.

`returnedID` The session ID returned by the DBGetConnInfo function.

`version` The version number of the data access extension assigned by the developer of the data access extension. It is not the same as the version number in the 'dflg' resource of the data access extension, which indicates the format of the data access extension.

`start` The time at which the session was opened, in ticks.

`host` The name of the remote system on which the data source is located.

`user` The name of the user who is establishing a session.

`password` The password associated with the user name.

`connStr` A connection string needed to establish a session.

`network` A string specifying the network in use for this session.

`buffer` A pointer to a buffer containing the item to be sent by the DBSend or DBSendItem functions or received by the DBGetItem function.

`err1` The primary error code returned by the data source.

`err2` The secondary error code returned by the data source.

`item1` A pointer to a NULL-terminated string that identifies the first object of the error message returned by the data source. The use of this parameter depends on the specific data source you are using.

`item2` A pointer to a NULL-terminated string that identifies the second object of the error message returned by the data source. The use of this parameter depends on the specific data source you are using.

`errorMsg` A pointer to the error message returned by the data source.

`timeout` The timeout period for the DBGetItem function, in sixtieths of seconds. When the data access extension executes the DBGetItem function, it requests a data item from the remote data source. If the remote data source does not return the requested data item in the amount of time specified by the timeout parameter, the data access extension should cancel execution of the DBGetItem function. The timeout value cannot be used if the DBGetItem function is called asynchronously.

`dataType` The data type of a requested or returned data item. Data types are described in "Getting Query Results" in the Data Access Manager chapter of *Inside Macintosh* Volume VI.

`sessNum` The session number. This number is assigned by the data access extension and is unique for all current sessions for a single data access extension only. The same session number can be assigned to concurrent sessions that use different data access extensions.

`state` The status of the data source.

Value	**Status**
noErr	Execution of a query successful; ready for another
rcDBValue	Output data is available
rcDBError	Execution of a query ended in an error
rcDBExec	Currently executing a query

`len` The length of the data item requested or returned.

`places` The number of decimal places in the data item.

`flags` Flags returned by the DBGetItem function or sent to the DBSendItem function. For the DBGetItem function, if the flags field is set to kDBLastColFlag (that is, the least significant bit is set to 1), the data item is in the last column of the row.

There are no flags currently defined for the DBSendItem function.

`abort` A parameter used by the DBBreak function. The meaning of this parameter depends on the specific implementation of the data source communications system you are using.

Data Access Extension Messages

There are sixteen values that the Data Access Manager can pass to the data access extension in the messages field of the data access extension parameter block. Thirteen of them correspond exactly to the thirteen low-level functions. The other three are used by the Data Access Manager to initialize and close the data access extension and to allow the data access extension to perform routine periodic tasks. The messages that correspond to low-level routines are not described in this section. Instead, only the parameters they use and the result codes they must be able to return are listed. For descriptions of these routines, see the section "Data Access Manager Routines" in the Data Access Manager chapter of *Inside Macintosh* Volume VI. The `DBOpen`, `DBClose`, and `DBIdle` messages are described in detail in this section.

Each parameter in the list is preceded by an arrow that indicates how the parameter is used, as follows:

→ The Data Access Manager passes the value of the parameter as input to the data access extension.

← The data access extension returns the value of the parameter after the routine has completed execution.

↔ The Data Access Manager provides a value for the parameter, and the data access extension returns another value.

DBOpen

Parameter block

→	00	message	word	routine selector; kDBOpen
←	02	ddevStorage	long	storage for data access extension

When an application calls the DBInit function or the DBStartQuery function (which calls the DBInit function), it specifies a data access extension. If that data access extension is not already in memory, the DBInit function loads it into memory and sends it the kDBOpen message. The data access extension should allocate any memory it needs at this time. Because the data access extension can be called by more than one application, it should allocate memory in the system heap rather than the application heap. The data access extension can also return a value in the ddevStorage field of the data access extension parameter block.

When the Data Access Manager calls the data access extension, the current resource file is the data access extension file and the default directory is the Extensions Folder on the current startup disk. The data access extension must ensure that both of these values are unchanged when it returns control to the Data Access Manager.

Result codes

noErr	0	Data Access Extension initialized successfully
rcDBError–	–802	Error initializing data access extension

DBClose

Parameter block

→	00	message	word	routine selector; kDBClose
→	02	ddevStorage	long	storage for data access extension

When an application calls the DBEnd function, closing the last open session for a data access extension, the Data Access Manager follows the kDBEnd message with a kDBClose message before removing the data access extension from memory. The data access extension should free any memory that it allocated before returning control to the Data Access Manager.

Result code

noErr	0	No error

DBIdle

Parameter block

→	00	message	word	routine selector; kDBIdle
↔	02	ddevStorage	long	storage for data access extension

The Data Access Manager periodically sends the kDBIdle message to each data access extension. The data access extension can ignore this message or take the opportunity to perform periodic tasks. Because the timing of the kDBIdle messages might not be regular, the data access extension must not depend on receiving these messages at particular times or with a particular frequency.

Result code

noErr	0	No error

DBInit

Parameter block

→	00	message	word	routine selector; kDBInit
↔	02	ddevStorage	long	storage for data access extension
→	06	asyncPB	long	pointer to asynch parameter block
←	10	sessID	long	session ID
→	26	host	long	pointer to name of remote system
→	30	user	long	pointer to user name
→	34	password	long	pointer to user password
→	38	connStr	long	pointer to connection string

The DBInit function initiates a session with a remote data source. See “Controlling the Session” in the Data Access Manager chapter of *Inside Macintosh* Volume VI for a complete description of this function.

Result codes

noErr	0	No error
rcDBError	–802	Error initiating session

DBEnd

Parameter block

→	00	message	word	routine selector; kDBEnd
↔	02	ddevStorage	long	storage for data access extension
→	06	asyncPB	long	pointer to asynch parameter block
→	10	sessID	long	session ID

The DBEnd function terminates a session with a remote data source and terminates the network connection between the application and the remote computer. See "Controlling the Session" in the Data Access Manager chapter of *Inside Macintosh* Volume VI for a complete description of this function.

Result codes

noErr	0	No error
rcDBError	–802	Error ending session

DBGetConnInfo

Parameter block

→	00	message	word	routine selector; kDBGetConnInfo
↔	02	ddevStorage	long	pointer to storage for ddev
→	06	asyncPB	long	pointer to asynch parameter block
→	10	sessID	long	session ID
←	14	returned ID	long	session ID returned
←	18	version	long	version number
←	22	start	long	session start time in ticks
←	26	host	long	pointer to name of remote system
←	30	user	long	pointer to user name
←	38	connStr	long	pointer to connection string
←	42	network	long	pointer to name of network
→	78	sessNum	word	session number
←	80	state	word	status of data source

The DBGetConnInfo function returns information about the specified session. See "Controlling the Session" in the Data Access Manager chapter of *Inside Macintosh* Volume VI for a complete description of this function.

Result codes

noErr	0	No error
rcDBBadSessNum	–808	Invalid session number

DBGetSessionNum

Parameter block

→	00	message	word	routine selector; kDBGetSessionNum
↔	02	ddevStorage	long	storage for data access extension
→	06	asyncPB	long	pointer to asynch parameter block

→	10	`sessID`	long	session ID
←	78	`sessNum`	word	session number

The `DBGetSessionNum` function returns a session number. See "Controlling the Session" in the Data Access Manager chapter of *Inside Macintosh* Volume VI for a complete description of this function.

Result codes

`noErr`	0	No error
`rcDBError`	– 802	Error getting session number

`DBKill`

Parameters used

→	00	`message`	word	routine selector; `kDBKill`
↔	02	`ddevStorage`	long	storage for data access extension
→	06	`asyncPB`	long	pointer to asynch parameter block

The `DBKill` function cancels the execution of an asynchronous call. See "Controlling the Session" in the Data Access Manager chapter of *Inside Macintosh* Volume VI for a complete description of this function.

Result codes

`noErr`	0	Asynchronous routine canceled successfully
`rcDBError`	–802	Error canceling routine

`DBSend`

Parameter block

→	00	`message`	word	routine selector; `kDBSend`
↔	02	`ddevStorage`	long	storage for data access extension
→	06	`asyncPB`	long	pointer to asynch parameter block
→	10	`sessID`	long	session ID
→	46	`buffer`	long	pointer to data buffer
→	82	`len`	word	length of data

The `DBSend` function sends a query or a portion of a query to the remote data source. See "Sending and Executing Queries" in the Data Access Manager chapter of *Inside Macintosh* Volume VI for a complete description of this function.

Result codes

noErr	0	No error
rcDBError	–802	Error trying to send text

DBSendItem

Parameter block

→	00	message	word	routine selector; kDBSendItem
↔	02	ddevStorage	long	storage for data access extension
→	06	asyncPB	long	pointer to asynch parameter block
→	10	sessID	long	session ID
→	46	buffer	long	pointer to data buffer
→	74	dataType	long	data type
→	82	len	word	length of data item
→	84	places	word	decimal places in data item
→	86	flags	word	flags

The DBSendItem function sends a single data item to the remote data source. See "Sending and Executing Queries" in the Data Access Manager chapter of *Inside Macintosh* Volume VI for a complete description of this function.

Result codes

noErr	0	No error
rcDBError	–802	Error trying to send item

DBExec

Parameter block

→	00	message	word	routine selector; kDBExec
↔	02	ddevStorage	long	storage for data access extension
→	06	asyncPB	long	pointer to asynch parameter block
→	10	sessID	long	session ID

The DBExec function initiates execution of a query. See "Sending and Executing Queries" in the Data Access Manager chapter of *Inside Macintosh* Volume VI for a complete description of this function.

Result codes

noErr	0	Execution has begun
rcDBError	–802	Error trying to begin execution

DBState

Parameter block

→	00	`message`	word	routine selector; `kDBState`
↔	02	`ddevStorage`	long	storage for data access extension
→	06	`asyncPB`	long	pointer to asynch parameter block
→	10	`sessID`	long	session ID

The result code returned by the `DBState` function indicates the status of the remote data source. See "Sending and Executing Queries" in the Data Access Manager chapter of *Inside Macintosh* Volume VI for a complete description of this function.

Result codes

`noErr`	0	No error; ready for more text
`rcDBValue`	–801	Output data available
`rcDBError`	– 802	Execution ended in an error
`rcDBExec`	–806	Currently executing query

DBGetErr

Parameter block

→	00	`message`	word	routine selector; `kDBGetErr`
↔	02	`ddevStorage`	long	storage for data access extension
→	06	`asyncPB`	long	pointer to asynch parameter block
→	10	`sessID`	long	session ID
←	50	`err1`	long	primary error code
←	54	`err2`	long	secondary error code
←	58	`item1`	long	pointer to first object of error message
←	62	`item2`	long	pointer to second object of error message
←	66	`errorMsg`	long	pointer to error message

The `DBGetErr` function retrieves error codes and error messages from a remote data source. See "Sending and Executing Queries" in the Data Access Manager chapter of *Inside Macintosh* Volume VI for a complete description of this function.

Result codes

`noErr`	0	No error
`rcDBError`	–802	Error retrieving error information

`DBBreak`

Parameter block

→	00	`message`	word	routine selector; `kDBBreak`
↔	02	`ddevStorage`	long	storage for data access extension
→	06	`asyncPB`	long	pointer to asynch parameter block
→	10	`sessID`	long	session ID
→	88	`abort`	byte	abort flag

The `DBBreak` function can halt execution of a query and reinitialize the remote data source, or it can unconditionally terminate a session with a data source. See "Sending and Executing Queries" in the Data Access Manager chapter of *Inside Macintosh* Volume VI for a complete description of this function.

Result codes

`noErr`	0	Execution has begun
`rcDBError–`	–802	Break or abort attempt was unsuccessful

`DBGetItem`

Parameter block

→	00	`message`	word	routine selector; `kDBGetItem`
↔	02	`ddevStorage`	long	storage for data access extension
→	06	`asyncPB`	long	pointer to asynch parameter block
→	10	`sessID`	long	session ID
→	46	`buffer`	long	pointer to data buffer
→	70	`timeout`	long	timeout value
↔	74	`dataType`	long	data type
↔	82	`len`	word	length of data item
↔	84	`places`	word	decimal places in data item
←	86	`flags`	word	flags

The `DBGetItem` function retrieves the next data item from the data source. See "Retrieving Results" in the Data Access Manager chapter of *Inside Macintosh* Volume VI for a complete description of this function.

Result codes

`noErr`	0	No error; no next data item
`rcDBValue`	–801	A nonzero data item was successfully retrieved
`rcDBNull`	–800	The data item was NULL
`rcDBError`	–802	No next data item; execution ended in an error

`rcDBBadType`	–803	Next data item not of requested data type
`rcDBBreak`	–805	Timed out

`DBUnGetItem`

Parameter block

→	00	`message`	word	routine selector; `kDBUnGetItem`
↔	02	`ddevStorage`	long	storage for data access extension
→	06	`asyncPB`	long	pointer to asynch parameter block
→	10	`sessID`	long	session ID

The `DBUnGetItem` function reverses the effect of the last call to the `DBGetItem` function. See "Retrieving Results" in the Data Access Manager chapter of *Inside Macintosh* Volume VI for a complete description of this function.

Result codes

`noErr`	0	No error
`rcDBError`	–802	Error executing function

Further Reference:

- *Inside Macintosh*, Volume VI, Data Access Manager chapter

FPSP Overview

The FPSP provides three basic emulation services for the 68040. First, it emulates many MC68881/2 instructions, including all transcendental functions and some arithmetic instructions. Second, the FPSP handles instructions that involve certain data classes (unnormalized and denormal floating-point numbers) or the packed decimal data format, which are not supported by the 68040 hardware. Finally, the FPSP provides exception handlers for certain floating-point exception conditions in order to emulate MC68881/2 behavior when user traps are either disabled or enabled. In the latter case, after completing its exception processing, the FPSP passes control to the user-provided handler.

On Quadra platforms executing MC68881/2 instructions, entry to the FPSP occurs automatically by trapping via one of several low-memory exception vectors, depending on which emulation service is required. The system installs the exception vector entries to the FPSP at boot time, and **applications should not tamper with these vectors.** Because the FPSP preempts the exception vectors for certain user-provided handlers in the MC68881/2 model, compatibility is a problem for old user code that contains floating-point exception handlers. Later sections will address the issues of compatibility in more detail.

Emulation of Unimplemented FPU Instructions

The following MC68881/2 arithmetic instructions are emulated by the FPSP, which produces results and exceptions identical to MC68881/2 platforms:

`FGETEXP`	Extract binary exponent of source
`FGETMAN`	Extract mantissa (significand) of source
`FINT`	Round source to integral value, using rounding mode in the FPCR
`FINTRZ`	Round source to integral value, using round-to-zero mode
`FMOD`	Modulo remainder of destination ÷ source with sign and lowest 7 bits of quotient delivered in FP status register (FPSR) quotient byte
`FMOVECR`	Move constant ROM to FP data register
`FREM`	IEEE remainder of destination ÷ source with sign and lowest 7 bits of quotient delivered in FPSR quotient byte
`FSCALE`	Scale (multiply) destination by 2^((int) source).

The following MC68881/2 transcendental functions are emulated by the FPSP:

`FACOS`	Inverse (arc) cosine (radians)
`FASIN`	Inverse (arc) sine (radians)
`FATAN`	Inverse (arc) tangent (radians)
`FATANH`	Inverse (arc) hyperbolic tangent
`FCOS`	Cosine of source in radians
`FCOSH`	Hyperbolic cosine
`FETOX`	Base e power (e^source)
`FETOXM1`	e^source - 1.0
`FLOG10`	Base 10 logarithm
`FLOG2`	Base 2 logarithm
`FLOGN`	Base e (natural) logarithm
`FLOGNP1`	Base e (natural) logarithm of (source + 1.0)

`FSIN`	Sine of source in radians
`FSINCOS`	Simultaneous sine and cosine (two destination registers)
`FSINH`	Hyperbolic sine
`FTAN`	Tangent of source in radians
`FTANH`	Hyperbolic tangent
`FTENTOX`	10.0^source
`FTWOTOX`	2.0^source

The algorithms used by the FPSP to calculate transcendental functions are both accurate and fast. Results will not always agree with those of the MC68881/2. When they disagree, the FPSP is generally more precise. The performance of the 68040 FPSP on transcendental functions is roughly equivalent to that of a similarly clocked MC68030/MC68882 combination.

When the 68040 in a Quadra attempts to execute any of the unimplemented MC68881/2 instructions, it traps, via vector number 11, the unimplemented F-Line opcode exception vector stored at vector offset (low-memory address) `$002C` to the FPSP. The corresponding exception handler in the FPSP saves the FPU state, decodes the instruction, fetches the operand(s), emulates the unimplemented instruction, and restores the appropriate state to the FPU. Operands involving unsupported data types or format are processed appropriately by this exception handler. To the user, the emulated instructions appear as atomic operations that produce valid results and that signal the proper floating-point exceptions. If an emulated instruction raises an enabled floating-point exception, program flow will vector to the appropriate user exception handler.

If the code executing in a Quadra contains an F-Line opcode that is undefined by the instruction sets of both the 68040 and MC68881/2, trapping to the FPSP via vector 11 also applies. In this case, the handler recognizes that no emulation is necessary, and it passes control to the system F-Line exception handler via a secondary vector stored in low memory.

Compatibility Note

If an application, such as a development or debugging environment, needs to install its own F-Line exception handler on Quadra platforms, it must **not** overwrite vector 11 at offset `$002C`. If it does, emulation of the unimplemented MC68881/2 instructions will be lost with disastrous effects to the executing program. Instead, the secondary F-Line exception vector, located at address **`$1FC8`**, should be used on Quadra platforms. As is the case on MC68881/2 platforms, the application should save the inherited F-Line exception vector (secondary vector in the case of Quadra platforms) and restore it upon termination or context switch.

Unimplemented Data Type/Format Support in the FPSP

The FPU in the 68040 does not support all of the floating-point data types and formats of the MC68881/2. The following data types require FPSP support:

> denormalized single (S), double (D), or extended (X) precision operand to an FPU instruction; and unnormalized X operand to an FPU instruction.

The following data format requires FPSP support:

> packed decimal real (P) format as source or destination for an FPU instruction.

When the 68040 encounters an unimplemented data type or format in the course of executing a hardware-supported FPU instruction, it traps, via exception vector 55, the FP unimplemented data type exception vector stored at vector offset (low-memory address) `$00DC` to the FPSP. Prior to the release of the 68040, this address was unassigned but reserved by Motorola. The unimplemented data type exception handler in the FPSP takes the appropriate action for the instruction and the exceptional operand or format.

For denormal S, denormal D, and all P format source operands, the FPSP converts the values to the normalized X equivalents, restores FPU state, and restarts the operation. If a source operand is an unnormalized X that can be converted to a normalized X, the instruction is also completed as described. If the instruction is a move out to P format in memory (`FMOVE.P FPn,<ea>`), the FPSP emulates the conversion from the extended source format to P format and writes the result to the effective address.

For denormal X operands or unnormalized X operands that reduce to denormal X values, the FPSP converts such operands to an internal normalized format that contains an extra exponent bit, restores state to the FPU, and restarts the operation if no exponent wrap condition will occur (for example, division of a denormal value by another denormal value). Otherwise, the FPSP emulates the entire instruction.

Denormalized values resulting from instructions executed by the 68040 hardware do not generate the unimplemented data type exception. Instead, a non-maskable underflow exception occurs which invokes a handler in the FPSP. This handler rounds the internal result appropriately according to the specified rounding precision and direction and delivers the result.

In the case of instructions that are emulated by the FPSP, the processing of unimplemented data type/format operands is handled within the confines of the emulation process. That is, the 68040 traps to the FPSP's unimplemented instruction handler, which is capable of recognizing and dealing with such operands.

Instructions, whether emulated or not, that use the P format as either source or destination have relatively poor performance because they require emulation of binary-to-decimal or decimal-to-binary conversions.

Idiosyncrasies

Binary operations (source and destination operands are both inputs) with P format source operands should avoid using FP1 as the destination operand because a bug in the FPSP causes spurious results in this case. If an unimplemented data type or format occurs as input to an operation, the exception is posted by the 68040 when the next FPU instruction is attempted. This deferred exception handling may appear not to deliver the correct result in a debugging environment that installs a breakpoint prior to the second FPU instruction.

FPSP Exception Handlers

Certain floating-point exception conditions on the 68040 require intervention by the FPSP in order to fix up results or other state. Some of the FPSP exception handlers are non-maskable in the sense that they are executed regardless of whether or not the exception is trap-enabled by the user. All of the FPSP floating-point exception handlers, whether non-maskable or not, are vectored via Motorola-designated locations in low-memory supervisor address space. If a user-enabled exception occurs, the FPSP exception handler is executed first before vectoring occurs to the user handler via a secondary vector maintained by the Quadra system. The user code must not modify the primary floating-point exception vectors to FPSP exception handlers. A later section will describe installation of user exception handlers.

The following is a brief description of FPSP exception handlers:

Branch/Set on Unordered (BSUN)

This maskable handler is invoked only if the user has enabled the BSUN exception. Entry to this handler is via vector number 48 stored at location `$00C0`. This handler updates the floating-point instruction address register (FPIAR) to contain the address of the floating-point branch/set instruction that generated the exception. It then invokes the user's handler via a secondary BSUN vector.

Inexact Result (INEX1/INEX2)

No FPSP handler is required. When enabled, INEX1 or INEX2 exceptions invoke the user's handler via vector number 49 at location `$00C4`.

Divide by Zero (DZ)

No FPSP handler is required. When enabled, the user's DZ handler is invoked via vector number 50 at location `$00C8`.

Underflow (UNFL)

This non-maskable handler is entered via vector number 51 at location `$00CC`. It determines and stores the properly rounded underflow result based upon the value of the intermediate result and the rounding precision/direction modes stored in the FPCR. If underflow is enabled in the FPCR, the user's handler is invoked via a secondary UNFL vector.

Operand Error (OPERR)

This non-maskable handler is entered via vector number 52 at location `$00D0`. It provides compatibility of results with the MC68881/2 for B, W, and L destination formats when the source operand is a NaN (Not-a-Number), infinity, or value too large for the integer format. If the OPERR exception is user-enabled, the FPSP handler invokes the user's handler via a secondary OPERR vector.

Overflow (OVFL)

This non-maskable handler is entered via vector number 53 at location `$00D4`. It determines and stores the properly rounded overflow result based on the value of the intermediate result and the rounding modes stored in the FPCR. If overflow is enabled in the FPCR, the user's handler is invoked via a secondary OVFL vector.

Signaling Not-a-Number (SNAN)

This non-maskable handler is entered via vector number 54 at location `$00D8`. It provides compatibility of results with the MC68881/2 for B, W, and L destination formats. If the SNAN exception is user-enabled, program flow is directed to the user's handler via a secondary vector.

If a program enables no floating-point exceptions in the FPCR, compatibility is not an issue. In this case, no user exception handlers need be installed. The program traps to non-maskable FPSP handlers as required for any fix-up of exceptional results or FPU state and then resumes execution.

Performance degradation by non-maskable FPSP floating-point exception handling is minimal in most cases because such intervention is rarely needed. The most common exception, INEX2, requires no FPSP support. Underflows and overflows are infrequent when the default extended rounding precision is employed. OPERR occurrences are also rare, unless many out-of-range conversions occur from floating-point to integer formats.

User Floating-Point Exception Handlers

Users who require floating-point exception handlers in their applications running on Quadra platforms must exercise some care in both the writing and the installation of such handlers. Moreover, if an application also targets Macintosh computers with MC68881/2 coprocessors and intends to resume processing via an `RTE` in an exception handler, its exception handlers must query which kind of FPU (MC68881/2 or 68040) is present and then execute hardware-specific code based on the query result. The reader is urged to consult the user manuals for the 68040 and MC68881/2 for details not covered by this Note.

Each floating-point exception on the 68040 is reported by either the conversion unit (CU) or normalization unit (NU) pipeline stage of the FPU. Exceptions reported by the CU are called E1 exceptions; they are detected relatively early in the execution of an FPU instruction. Exceptions reported by the NU are called E3 exceptions; they are detected late in the execution of FPU instructions as the NU attempts to normalize and round the intermediate result for storage in a destination FP register. E1 exceptions include all floating-point exception types. The only E3 exceptions are OVFL, UNFL, and INEX2 occurring on opclass 0 (register-to-register) and opclass 2 (memory-to-register) instructions. If both E3 and E1 exceptions exist at the same time, the E3 exception should be handled first, allowing the 68040 to subsequently trap to handle the pending E1 exception.

There are two `FSAVE` stack frames for floating-point exceptions on the 68040. E1 exceptions produce the unimplemented instruction FPU state frame, and E3 exceptions produce the busy FPU state frame. Both of these frames begin with a 1-byte version number followed by a 1-byte frame length. The version number for Quadra 68040s is $41. For this version of the 68040, the frame length for E1 exceptions is $30, making the unimplemented instruction FPU state frame 52 bytes in size (counting the 4-byte header). The busy frame for E3 exceptions has a frame length of $60 and total size of 100 bytes.

Both 68040 floating-point exception `FSAVE` stack frames contain information that may be of use to the user's exception handler. There are two 12-byte fields containing the source and destination operands in extended precision. There are two 3-bit tag fields which classify the source and destination operands as to whether they are normalized, denormalized, zero, infinite, or NaN.There are 2 bits (E1 and E3) which, if set, indicate which pipeline stage of the FPU (CU or NU) detected the pending exception(s). Both `FSAVE` frames encode the command word of the exceptional floating-point instruction, albeit in different fields.

As a minimum, user floating-point exception handlers on 68040 platforms must issue an `FSAVE` instruction as the first FPU operation, clear the exception state of the FPU, and resume processing via the `RTE` instruction. For E3 exceptions, the E3 bit in the `FSAVE` stack frame must be cleared and the `FRESTORE` instruction must be issued prior to the `RTE` instruction. For E1 exceptions, the minimum requirement is to throw away the `FSAVE` stack frame and to resume processing via RTE. Another method of clearing the exception state for E1 exceptions is to clear the E1 bit in the FSAVE stack frame and issue the FRESTORE prior to the RTE. The E1 and E3 bits are bits 2 and 1 (bit position 0 representing the least significant bit), respectively, of the byte which is located 28 bytes below the high-address end of either FSAVE frame.

Minimum Floating-Point Exception Handler for the MC68881/2 and Quadra

The following code sequence serves as a minimum handler for all enabled floating-point exceptions except BSUN on both with MC68881/2 platforms and Quadra computers. This handler simply clears the exceptional condition in the FPU and resumes execution without attempting to modify any other FPU state. A minimal BSUN handler would require additional intervention (via one of four methods outlined in the user manuals for the 68040 and the MC68881/2) to prevent infinite looping on the BSUN trap.

```
; ***********************************************************************
; Minimum user handler for enabled INEX, DZ, UNFL, OPERR, OVFL,
; or SNAN floating-point exception on either MC68881/2 or
; Macintosh Quadra platforms.
;
; NOTE:  For enabled DZ, OPERR, and SNAN exceptions for instructions
;        with FP register destinations, no result is delivered at all to the
;        destination register.
; ***********************************************************************
HANDLER:
            FSAVE          -(SP)          ; save internal FPU state
            MOVE.L         D0,-(SP)       ; save D0, STACK:  D0 save < FSAVE frame
            MOVEQ          #0,D0          ; zero D0
            MOVE.B         4(SP),D0       ; NULL frame?

            BEQ.B          @NULL          ; yes, restore D0 and FPU state

            CMPI.B         #$41,D0        ; Quadra (68040) ID?

            BNE.B          @COPROC        ; no, assume MC68881/2

; Quadra FSAVE frame
            MOVE.B         5(SP),D0       ; D0 <- frame size

            BEQ.B          @NULL          ; restore state if 68040 IDLE frame

; Quadra UNIMPLEMENTED INSTRUCTION or BUSY FSAVE frame
            SUBI.B         #20,D0         ; D0 <- offset of E1/E3 byte from (SP)
            BCLR.B         #1,(SP,D0)     ; test and clear E3 byte
            BNE.B          @NULL          ; restore state if E3 was set

            BCLR.B         #2,(SP,D0)     ; E1 exception, clear E1 byte

; Restore state and resume execution
@NULL:      MOVE.L         (SP)+,D0       ; restore D0, STACK:  FSAVE frame
            FRESTORE       (SP)+          ; restore FPU state
            RTE                           ; resume processing
```

```
; MC68881/2 IDLE FSAVE frame
@COPROC:     MOVE.B        5(SP),D0        ; D0 <- IDLE frame size
             ADDQ.B        #4,D0           ; compensate for D0 save value on stack
             BSET.B        #3,(SP,D0)      ; set bit 27 of BIU
             BRA.B         @NULL           ; restore state
```

Installation of User Floating-Point Exception Handlers

Current MPW language libraries (MPW 2.0.2 or later releases and Language Systems FORTRAN version 3.0) provide for the vectoring of user floating-point exception handlers in a consistent and portable fashion for both Quadra and MC68881/2 Macintosh platforms. The C functions settrapvector and gettrapvector, the Pascal procedures SetTrapVector and GetTrapVector, and the Language Systems FORTRAN subroutines SetTrapVector and GetTrapVector allow users to install and read vectors to their floating-point exception handlers via the use of the TrapVector structure. The relevant interface files for these operations are {CIncludes}SANE.h, {PInterfaces}SANE.p, and {FIncludes}SANE.f.

A TrapVector structure is composed of seven 4-byte fields that represent the entry-point addresses of the user's BSUN, INEX, DZ, UNFL, OPERR, OVFL, and SNAN exception handlers, respectively. GetTrapVector routines read the current floating-point exception vectors into a TrapVector structure. In order to install their own exception handlers, users must first initialize a TrapVector structure with entry points of their handler routines and then invoke a SetTrapVector routine with that structure as the operand.

GetTrapVector and SetTrapVector routines involve privileged operations because they access Motorola low-memory vector table locations. For Quadra platforms, the situation is further complicated by the fact that five of the seven user floating-point exception vectors are stored by the system in secondary locations because the FPSP has preempted the original vector table locations. GetTrapVector and SetTrapVector implementations circumvent these difficulties by calling a system utility, **PrivTrap**, which does all of the work of querying or installing the user's vectors.

The PrivTrap Mechanism

PrivTrap is implemented as a system trap, **$A097**. Upon entry, it expects a selector value in register D0.W and a TrapVector structure address in address register A0. The GetTrapVector operation requires a selector value of 1; in this case, PrivTrap reads the current floating-point exception vectors into the TrapVector structure at (A0). The selector value of 2 invokes the SetTrapVector operation; the user's exception vectors in the TrapVector structure at (A0) are installed appropriately in the system. In either case, registers A0 and A1 are modified upon exit.

As of the drafting of this Note, only the Quadra and Powerbook 170 platforms running System 7.0.1 have the PrivTrap mechanism built into their systems. Individual MPW library functions that require PrivTrap functionality first query if PrivTrap is installed. If it is not, the library routines will install and call a version of the trap appropriate for an MC68881/2 platform.

Implementation Notes

Since MultiFinder under System 6.0.x and Finder under current versions of System 7 do not include user exception vectors among the FPU state which is saved and restored at context switch, it is the responsibility of an application that enables floating-point exceptions to save inherited user exception vectors and to restore them upon termination or context switch. The inherited vectors may be read using the `GetTrapVector` operation. The application installs its floating-point exception handlers via the `SetTrapVector` operation. At context switch or program termination, `SetTrapVector` should be used to restore the appropriate exception vectors. If the above regimen is followed, the application's `TrapVector` structure may contain arbitrary values for vectors corresponding to disabled exceptions.

Performance Issues

In order to extract the maximum floating-point performance on a Quadra, an application should avoid invoking emulation by the FPSP whenever possible. Unfortunately, FPU instruction sequences that optimize Quadra performance often degrade performance to some extent on MC68881/2 platforms. Programmers must always weigh the performance requirements of their various target platforms when writing floating-point code.

Transcendental Functions

Although all FPU transcendental function instructions are emulated by the FPSP on Quadra platforms, performance is comparable to a similarly clocked platform using the MC68882.

Unimplemented Arithmetic Functions

If deemed desirable for performance reasons on Quadra platforms, workarounds can readily be devised for most of the arithmetic FPU instructions that are emulated by the FPSP. The **FMOD** and **FREM** instructions are the notable exceptions since they involve an iterative algorithm in their most general cases. The functionality of the remaining unimplemented arithmetic instructions can be emulated as follows:

> **FGETEXP** If the argument is a NaN or zero, return the argument. If the argument is infinite, return a NaN and signal OPERR. Otherwise, write the floating-point argument to stack, extract, and unbias the exponent using integer operations, and deliver the result to FPn using `FMOVE.L <ea>,FPn`.
>
> **FGETMAN** If the argument is a NaN or zero, return the argument. If the argument is infinite, return a NaN and signal OPERR. Otherwise, write the floating-point argument to the stack in extended format, normalize the significand (mantissa) if necessary, set the exponent bits to $3FFF, retain the original sign bit, and deliver the result to FPn using `FMOVE.X <ea>,FPn`.
>
> **FINT** If the argument is zero or if the exponent of the argument is greater than 62, return the argument. If the exponent of the argument is less than 31, round the argument to integral value by conversion to integer format via `FMOVE.L FPn,<ea>` followed by conversion back to X format via `FMOVE.L <ea>,FPm`. Otherwise, decompose the argument into an integral part (via integer operations on the X format on the stack) and a fractional part (via subtraction of the integral part from the argument), convert the fractional part to an integer via `FMOVE.L FPn,<ea>`, and add the integer to the integral part.

FINTRZ Using integer operations on the argument stored in extended format on the stack, test and zero out the fractional part. Set INEX2 if any fraction bits were nonzero. The test for inexactness may be omitted if the application is indifferent to INEX2 being signaled by this rounding operation.

FMOVECR Store desired constant in extended format in the code segment of program and load it via `FMOVE.X <ea>,FPn.`

FSCALE Convert the integral source operand n to a floating-point factor 2.0^n on the stack. Obtain the scale result via multiplication of that factor with the destination operand.

FINTRZ and Floating-Point → Integer Conversions

The most common compiler-generated unimplemented arithmetic FPU instruction is `FINTRZ` during conversions of floating-point values to various signed integer formats in C or FORTRAN source code. For example, to convert the value in `FPn` to 32-bit integer value at <ea>, a compiler will generate the following code sequence:

```
FINTRZ          FPn,FPm                 ; truncate to integral value
FMOVE.L         FPm,<ea>                ; convert to integral format
```

If the application is running in (IEEE 754) default mode (FPCR = $00000000: no exceptions are enabled, rounding precision is extended, rounding direction is round-to-nearest), the following code sequence will accomplish the same conversion with optimal performance on a Quadra and with minimal performance degradation on an MC68881/2 platform:

```
FMOVE.L         #$00000010,FPCR         ; set round-to-zero mode
FMOVE.L         FPn,<ea>                ; truncate to integral format
FMOVE.L         #$00000000,FPCR         ; restore default modes
```

If the user's FPCR setting is not the default, the last sequence must be modified to save and restore the user's FPCR setting at the cost of several instructions and some temporary storage. Throughput for these conversions may be enhanced if the application requires an array of floating-point values to be converted, because the FPCR needs to be modified only once before and once after all conversions are done via the `FMOVE.L FPn,<ea>` step. Out-of-range source values result in degraded performance on Quadra computers due to nonmaskable vectoring to the OPERR handler in the FPSP.

Workarounds for conversions from floating-point values to the unsigned integer formats of C are more complicated and of necessity slower than those to signed integer formats.

Miscellaneous Performance Tips for Quadra Applications

In order to minimize trapping to the FPSP for handling of exceptional conditions, data types, or data formats, the following hints may prove useful:

- Applications should run with extended rounding precision set in the FPCR.
- Temporary storage for intermediate floating-point results should be in extended format and preferably in FP registers.
- Applications should avoid the generation of unnormalized extended format values via integer operations with subsequent reliance on the FPU to normalize the results.

- Applications should avoid the extensive use of the Motorola packed decimal (P) data format.

MPW QR6 Libraries

The MPW QR6 folder in the E.T.O. #6 Developers CD contains C and Pascal libraries that have been performance-tuned. In particular, some of the `-mc68881` mode implementations have been modified to obtain better performance on Quadra platforms. Included among the new implementations are conversions from floating-point to the unsigned integer formats of C. Unfortunately, conversions to signed integer formats are generated in-line by the C compiler and thus still include the `FINTRZ` instruction, which is emulated by the FPSP in Quadra platforms.

Summary

FPU operations on Quadra platforms are performed by a combination of circuitry in the 68040 microprocessor and emulation code in the FPSP. The 68040 provides very fast implementations of most of the basic floating-point arithmetic functions in the MC68881/2 instruction set. The FPSP emulates all transcendental functions and some arithmetic functions. In addition, the FPSP handles instructions that involve certain data types/formats that are unsupported by the 68040 hardware and fixes up state when certain exceptional conditions arise during processing.

Compatibility of results relative to MC68881/2 platforms holds for all FPU arithmetic instructions, whether or not they are emulated on Quadra computers. Results for transcendental FPU instructions may differ, and they are generally more precise on the Quadra.

FPU applications that run with no floating-point exceptions enabled in the FPCR and that do not install an unimplemented F-Line Opcode handler will run without modification on both MC68881/2 and Quadra platforms. User unimplemented F-Line exception handlers are installed via vector 11 at address `$002C` on MC68881/2 platforms and via a secondary vector at address `$1FC8` on Quadra platforms. Similarly, installation of user floating-point exception handlers for enabled exceptions must take care not to overwrite entry points to the FPSP on Quadra platforms. MPW libraries provide high-level installation procedures for user floating-point exception handlers. If such handlers are to run on all FPU platforms, they must take into account the differences in `FSAVE` state frames for Quadra and MC68881/2 platforms.

Optimizing FPU performance on Quadra computers is largely a matter of understanding the conditions under which the FPSP is invoked and then avoiding such conditions via workarounds whenever possible. Code sequences thus optimized for Quadra computers will often provide less than optimal performance on MC68881/2 platforms.

Further Reference:

- *MC68881/MC68882 Floating-Point Coprocessor User's Manual*
- *MC68040 32-Bit Microprocessor User's Manual*
- *MC68040 Designer's Manual,* Section 3: Floating-Point Emulation
- *M68000 Family Programmer's Reference Manual*
- *IEEE Standard for Binary Floating-Point Arithmetic* (ANSI/IEEE Std 754-1985)

#318: Serial PollProc

Written by: Rich "I See Colors" Collyer & Dave Wong June 1992

This Technical Note discusses how to make a PollProc for your MIDI (Musical Instrument Digital Interface) driver on the Macintosh PowerBook 140 and 170.

For MIDI Consumption Only

You are writing your own MIDI driver and your driver does not fully work on the PowerBook 170/140. The PollProc support that might help solve your problem has been undocumented until now because it has a bug in it which if ever fixed would cause major problems with every PollProc ever made. The bug is in the way that the PollProc mechanism handles data errors - it doesn't. At some point this might get fixed and an fix would require changes to any existing PollProcs. We are only documenting this now because we (Apple) would like to see high MIDI data transfers working on the PowerBook 170/140, and the PollProc support is the only solution we have been able to come up with. If you do use this information, be aware that the PollProc mechanism may change in the future and when it does your PollProc will need to change. We do not recommend the use of PollProc mechanisms on any other Macintosh computers.

What the Problem Is

When doing a large data dump, such as downloading MIDI instruments or sampled sounds, MIDI data overruns the input serial port on the PowerBook 170/140.

Background

• MIDI developers and users have been reporting problems that occur on PowerBook 170/140. The specific problem described by the developers is that data overrun errors occur (that is, serial data is lost). MIDI data is serial data that is transmitted at 31.25 Kbaud. This means that one byte of data is transmitted approximately every 200 usec.

• The serial port has a three-byte FIFO, which means that three bytes of data could be stored temporarily before a data overrun (data loss).

• The MIDI functioned OK on the original portable, but just barely. The overhead of communicating to the power manager microprocessor seems to interfere with MIDI.

• The 170/140 hardware required certain changes to the power management software because of changes in the hardware. In particular, the hardware changes required changes to the protocol used to communicate to the power manager microprocessor.

• The 170/140 has software backlight controls that cause constant communication between the 68030 and the power manager microprocessor (every 200 msec).

Findings

• The MIDI driver loses data. The 170/140 has a real-time problem and is not able to keep up with sustained MIDI data rates. The culprit is the communication between the 68030 and the power manager microprocessor. During this communication, interrupts must be disabled for a certain amount of time.

• On the 170/140, the protocol for this communication was changed from that of the portable. The 170/140 can cause interrupt blackouts up to 6 msec as compared to approximately 500-700 usec on the portable (estimation only). Assuming the worst case, during the 6 msec blackout as many as 30 MIDI data bytes could have been sent. Since the FIFO on the serial port is only 3 deep, this means that as many as 27 bytes could have been lost (remember these are ballpark figures only).

• The problem is aggravated by increased power manager communications for backlight controls.

What the Solution Is

• Changing the protocol to the power manager microprocessor (given the hardware constraints) is not practical since the problem is not completely solved and could cause other system problems.

• At the moment, no Apple-only solution is possible.

• A developer-only solution is possible. Currently an internal mechanism exists to keep up with high data rates on the modem port. This mechanism, called PollProc (Polling Procedure), will allow the ROM code to handle the serial port during known interrupt blackout windows, which helps prevent data loss. The power manager communication software currently checks for such a routine and will use it automatically if it is present. In addition to correcting this problem, this will also allow MIDI to perform during floppy activity (which has similar real-time problems) since the floppy driver also checks for PollProc.

In the code which is included at the end of this Tech Note, there is a extra Procedure which is call ProcessByte. In the sample this routine does nothing. The reason for the sample not doing anything is due to the nature of the routine. What the routine does is completely dependent on what the serial driver is doing or wants to do with the data as it is read into the machine. This routine might be used to decompress data, compress data, decoded the data, or do any other kind of alteration you wish to do to it. The Macintosh OS does not do anything to the data, so this routine is not needed, but your application might need this routine - it is up to you, just don't do to much at this time. It is important to remember that you need to get in and out of the PollProc as fast as possible.

What Is a PollProc and How Does It Work?

A PollProc is a routine that a serial driver implements so it can still get data when the OS turns interrupts off for a significant amount of time. Although PollProc mechanisms work for generic serial drivers, it is recommended that you use this feature in your MIDI driver only on the PowerBook 170/140. When the MIDI driver is opened and supports PollProc mechanisms, it needs to place a pointer to this routine in the low-memory global—PollProc. When the OS (such as the Power Manager and floppy driver) turns off interrupts, it checks to see if the low-memory global is nil or not. If the global is not nil, then it the OS will poll the SCC for incoming data and stuff the data into a buffer. Then just before the OS turns the interrupts back on, it calls the

PollProc and passes the buffer to it. The PollProc will be able to handle the data as if it were coming in via the serial port.

The PollProc is supported only on port A. Port B PollProcs are not supported.

The comments in the following code give more detail about how to implement the PollProc.

PollProc Sample Code

```
;_____________________________________________________________________________
;InputPollData  -  process SCC input data
;
;This routine is called via the low-memory vector PollProc by system code
;that had interrupts disabled for a long enough period of time that SCC
;data may have been lost. The system code will poll the SCC for data during
;the time it had interrupts disabled and call this routine right before
;interrupts are reenabled. The address of the InputPollData routine should
;be written into the "PollProc" low-memory vector when the SCC channel A
;driver is opened. The "PollProc" low-memory vector should be zeroed
;when the driver is closed.
;
;The InputPollData routine will be called with data to be processed on the
;stack. This routine should process the data as if it had been received by
;the driver's receive data interrupt routine.
;
;Note:          PollProc mechanisms are not necessary on SCC IOP based machines and should
               not be used.
;
;Input:         a6.l  =  SCC channel A data pointer
;Output:        none
;
;allowed to trash:      a0-a1/a3-a4

PollStack      equ    $13A    ; SCC poll data start stack location [pointer]
PollProc       equ    $13E    ; SCC poll data procedure [pointer]
RxCA           equ    0       ; Bit zero of SCC RR0 indicates receive char avail.

InputPollData

movea.l        (sp)+,a4             ; Save return address.
movea.l        PollStack,a3         ; a3 = ptr to beginning of data on stack.

;First empty all the data from the SCC. This may not be needed, but it is
;here for completeness. The drivers that will use the PollProc mechanism
;will already have similar code to this, so whether you implement this or
;not is more of a personal call. Our recommendation is that you try to go
;without the code, and if you find you do need it, then implement it.

@EmptySCC
movea.l        SCCRd,a0             ; base addr of SCC read register 0 from low mem
addq.w         #2,a0                ; Add offset to get to channel A registers.
btst.b         #RxCA,(a0)           ; Test if SCC data is available.
beq.s          @ProcessData         ; no additional SCC data
move.b         (a6),-(sp)           ; Move SCC channel A data onto stack.
bra.s          @EmptySCC
```

```
;Process all the SCC data on the stack as if it were read in normally by
;the SCC driver's receive interrupt routine. There is stack data starting
;from the address in the low-mem PollStack, to the current stack pointer.

@ProcessData
cmp.l         sp,a3                 ; Have we processed all the stack data?
beq.s         @Done                 ; We are done.
subq.w        #1,a3                 ; Skip over garbage byte because stack pushes words.
move.b        (a3)+,d0              ; Get the saved data byte.

bsr.s         ProcessByte           ; Call driver routine to process the data byte.
bra.s         @EmptySCC             ; Check for SCC data before processing next saved byte.

;Done - cleanup stack of saved data

@Done
move.l        PollStack,sp          ; Set stack ptr to pop saved data.
jmp           (a4)                  ; Jump to the return address.



;_______________________________________________________________________________
;ProcessByte  -  process saved SCC input data
;
;This routine is a stub example routine that will process a saved data
;byte as if the driver had read in the byte normally.
;
;Input:        d0.b  =  SCC channel A data byte
;Output:       none
;

ProcessByte

;Fill in necessary code.

rts
```

THE SOUND MANAGER

See Also
Macintosh Technical Notes #168, and #208
Audio Interchange File Format specification
Macintosh Audio Compression/Expansion specification

Written By Jim Reekes October 2nd, 1988

SOUND ADVICE

This document describes the System 6.0.2 Sound Manager. The original chapter describing the Sound Manager is ambiguous, inaccurate, and often contradicts itself. This chapter hopefully will clear up the confusion and get developers using the Sound Manager as was originally intended. This document replaces the Sound Manager chapter originally published in *Inside Macintosh Volume V*.

The Sound Manager is a replacement for the older Sound Driver documented in Inside Macintosh Volume II. The abilities of the Sound Driver are currently supported by the Sound Manager and it will utilize future hardware improvements. The Sound Manager offers more flexible ways of doing things and, includes new features and options, all requiring less programming effort. Many applications do not require the use of sound and therefore do not need to be concerned with the Sound Manager. Refer to the *Human Interface Guideline: The Apple Desktop Interface* when using sound.

A fundamental knowledge of music and sound synthesis is presumed in this document. There are utilities available from third parties that aid in the development of creating sampled sound resources. Creating wave table data or discussing the abilities of wave synthesis versus sampled sound synthesis is not covered in this document. Two good reference books are *Computer Music, Synthesis, Composition, and Performance* by Charles Dodge and Thomas A. Jerse, and *Principles of Digital Audio* by Ken Pohlman.

This document contains an overview of the Sound Manager, and a detailed description of sound resources, routines and commands. All of the known bugs and limitations are collected into one section, "The Current Sound Manager". A bug icon is used to point out information contained in this section that is relative to the text being read. For example, when reading about a sound command if a bug icon is shown, make sure you have read the "Current Sound Manager" section regarding that command.

TABLE OF CONTENTS

INTRODUCTION

The Sound Manager is a collection of routines that can be used to create sounds without knowledge of, or dependence on, the hardware available. By using the Sound Manager, applications are assured of upward-compatibility with future hardware and software releases. The Sound Manager will always take advantage of hardware advancements. Applications using the Sound Manager now will gain those advantages. When a command is sent to the Sound Manager, it is really a request. For example, if sound code written to play on a Macintosh II is being used on a Macintosh Plus or Macintosh SE (which have slower CPU clocks and less capable audio hardware) the Sound Manager will use synthesizers fitted best to those machine's abilities. Conversely, future Macintoshes may have improved audio hardware, and that same code will be utilized by the Sound Manager to take full advantage of these as-yet-undetermined hardwares. All of this is transparent to the application, yet serves to make that application compatible with the full line of Macintosh computers, present and future.

A *synthesizer* is very similar to a device driver. A synthesizer is the code responsible for interpreting the most general sound commands and using the hardware available to produce it. A synthesizer is stored as a resource which the Sound Manager will install. Customized synthesizers are supplied for every Macintosh configuration. Only one synthesizer can be active at any time. Apple's sound hardware is only supported when used with Apple's synthesizers. Writing synthesizers for Apple's hardware is not supported. Writing custom synthesizers for non-Apple hardware is beyond the scope of this document. All references to synthesizers in this document pertain to the Apple synthesizers that are supplied with the Sound Manager.

Modifiers are used to perform pre-processing of commands before they are received by a synthesizer. Modifiers can ignore, alter, remove, or add commands, or perform periodic functions. A modifier is a procedure in memory, or a resource which the Sound Manager can install. For example, if the application wanted to play a melody transposed up by an octave a modifier could be used to replace notes with notes that are an octave higher.

Instructions for a synthesizer and modifier are sent through a command queue called a *sound channel*. Sound channels provide a means of linking applications to the audio hardware. The application provides a sequence of commands which are processed through a number of modifiers (if any) and finally through a synthesizer that creates the sound with the hardware.

USING THE SOUND MANAGER

The Sound Manager code that runs on the Macintosh Plus is the same that is used on the Macintosh SE. The code running on the Macintosh II is different, since it has the Apple Sound Chip installed. The Apple Sound Chip was developed to reduce the CPU's involvement with producing sound and to extend the capabilities of the Sound Manager.

⚠ The Sound Manager requires the use of the VIA1 timer T1. This conflicts with some third party MIDI drivers. As such, it is not possible to use both the Sound Manager and these MIDI applications.

There are two types of resources used by the Sound Manager, `'snd '` and `'snth'`. A `'snd '` resource contains data and/or commands. A `'snth'` resource is code used as a synthesizer or modifier to interpret the commands sent into a channel. Generally, applications only need to be concerned with `'snd '` resources. More information on the formats of `'snd '` resources and their use is given later.

The Sound Manager provides a range of methods for creating sound on the Macintosh. Most applications will only need to use a few of the Sound Manager routines. At the simplest end of the range is the use of the note synthesizer to play a simple melody or `_SndPlay`. `_SndPlay` only requires a proper `'snd '` resource. Such a resource will contain the necessary information to create a channel linked to the required synthesizer and the commands to be sent into that channel. An application can use the following code to create a sound with this method:

```
myChan := NIL;
sndHandle := GetNamedResource ('snd ', 'myBeep');
myErr := SndPlay (myChan, sndHandle, FALSE);
```

For more complete control of the sound channel, an application can open a sound channel with `_SndNewChannel`. The application will then send commands to that channel with `_SndDoCommand` or `_SndDoImmediate`. When the application's sound is completed, the application closes the channel with `_SndDisposeChannel`.

The System Beep

The trap `_SysBeep` is a call to the Sound Manager. The sound of the System Beep is selected by the user in the Control Panel using the Sound `'cdev'`. Except for the "Simple Beep", `_SysBeep` will be performed by the Sound Manager. If this sound is selected on a Macintosh that doesn't have the Apple Sound Chip (i.e. the Macintosh Plus and SE), the beep will be generated by the original ROM code. This has the benefit of bypassing the Sound Manager and the potential conflict of third party MIDI drivers which both use the VIA1 timer T1. Thus, this conflict over the timer can be avoided by setting the System beep to the "Simple Beep" using the Sound `'cdev'` in the Control Panel.

If an application has an active synthesizer, then `_SysBeep` may not generate any sound. This is because only one synthesizer can be active at any time. On a Macintosh without the Apple Sound Chip (i.e. the Plus and SE) when the "Simple Beep" is selected the beep will be heard, since it bypasses the Sound Manager. Applications should dispose of their channels as soon as they have completed making sound, allowing the `_SysBeep` to be heard.

⚠ `_SysBeep` cannot be called at interrupt time since the Sound Manager will attempt to allocate memory and load a resource.

🐛 Refer to the section "Current Sound Manager" regarding `_SysBeep` on a Macintosh Plus and SE.

The Note Synthesizer

The note synthesizer is the simplest of all the synthesizers supplied with the Sound Manager. The sound produced by this synthesizer is based upon a square wave. An application cannot play back a wave form description or recorded sound when using this synthesizer. Very little set up is required to use this synthesizer. It also has the advantage of using little CPU time. It can be used for creating simple monophonic melodies.

The Wave Table Synthesizer

The wave table synthesizer will produce sounds based on a description of a single wave cycle. This cycle is called a wave table and is represented as an array of bytes describing the timbre (tone) of a sound. Applications may use any number of bytes to represent the wave, but 512 is the recommended length since *the Sound Manager will re-sample it to this length.* A wave table can be pulled in from a resource or computed by the application at run time. To install a wave table in a channel, use the `waveTableCmd`. Up to four wave table channels can be opened at once allowing an application to play chords, melodies with harmonies and polyphonic melodies.

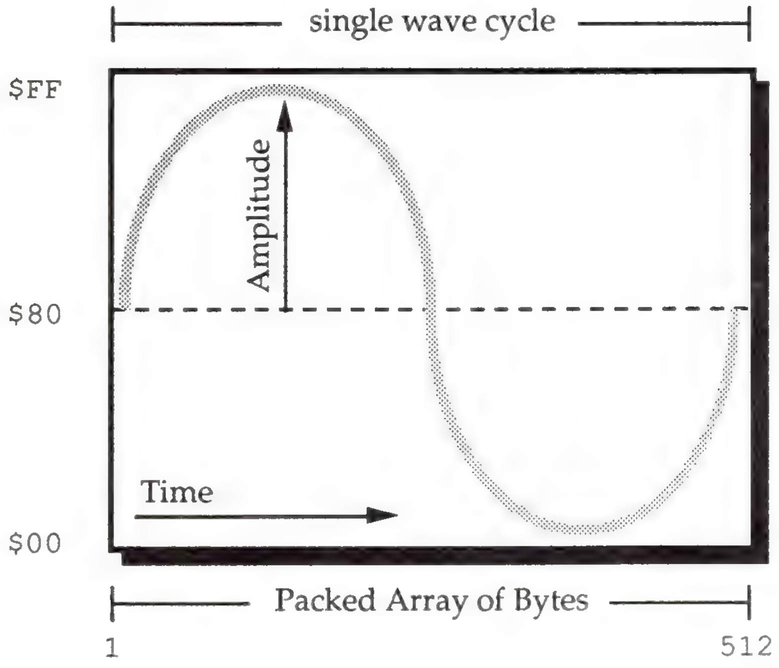


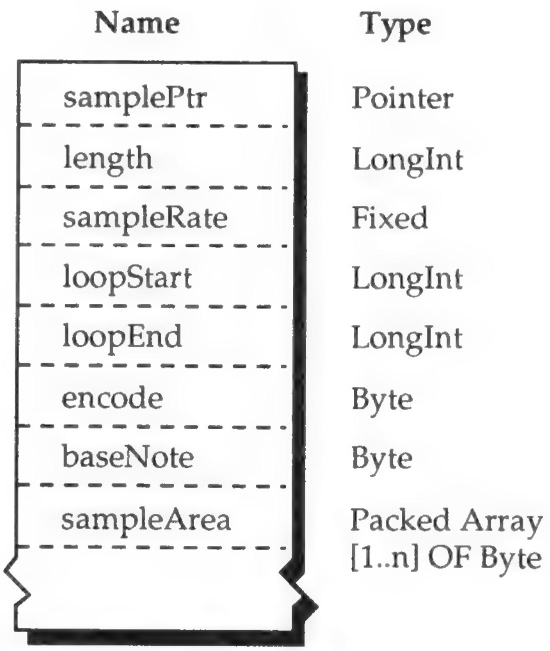
Figure 1 Graph of a Wave Table

A wave table is a sequence of wave amplitudes measured at fixed intervals. Figure 1 represents a sine wave being converted into a wave table by taking the value of the wave's amplitude at every 1/512th interval. A wave table is represented as a `PACKED ARRAY [1..512] OF BYTE`. Each byte may contain the value of `$00` through `$FF` inclusive. These bytes are considered offset values where `$80` represents a zero level of amplitude, `$00` is the largest negative value, and `$FF` is the largest positive value. The wave table synthesizer loops through the wave table for the duration of the sound.

Refer to the section "Current Sound Manager" regarding the wave table synthesizer on the Macintosh Plus and SE.

The Sampled Sound Synthesizer

The sampled sound synthesizer will play back digitally recorded (or computed) sounds. These sampled sounds are passed to the synthesizer in the form of a sampled sound header. This header can be played at the original sample rate, or at other rates to change its pitch. The sampled sound can be installed into a channel and then used as an instrument to play a sequence of notes. Thus a sampled sound, such as a harpsichord, can be used to play a melody. This synthesizer is typically used with pre-recorded sounds such as speech, songs or special effects. Developers concerned with saving sampled sound files need to refer to the Audio Interchange File Format available from the Apple Programmer's and Developer's Association. Figure 2 shows the structure of the sampled sound header used by the sampled sound synthesizer.



Name	Type
samplePtr	Pointer
length	LongInt
sampleRate	Fixed
loopStart	LongInt
loopEnd	LongInt
encode	Byte
baseNote	Byte
sampleArea	Packed Array [1..n] OF Byte

Figure 2 Sampled Sound Header

The first field of a sampled sound header is a `POINTER`. If the sampled sound is located immediately in memory after the `baseNote`, this field is `NIL`, otherwise it will be a pointer to the sample sound data. The `length` field is the number of bytes in the `PACKED ARRAY [1..n] OF BYTE` containing the sampled sound, `n` being this `length`.

RATE	DECIMAL	HEX
5kHz	5563.6363	$15BB.A2E8
7kHz	7418.1818	$1CFA.2E8B
11kHz	11127.2727	$2B77.45D1
22kHz	22254.5454	$56EE.8BA3
44kHz	44100.0000	$AC44.0000

Table 1 Sample Rates

The sampleRate is the rate at which the sample was originally recorded. These unsigned numbers are of type FIXED. The approximate sample rates are shown in Table 1.

The loop points contained within the sample header specifies the portion of the sample to be used by the Sound Manager when determining the duration of a noteCmd. These loop points specify the byte numbers in the sampled data used as the beginning and ending points to cycle through while playing the sound.

Refer to the section "Current Sound Manager" regarding the noteCmd and looping with a sampled sound header.

The encode option is used to determine the method of encoding used in the sample. The current encode options are shown below.

```
stdSH = $00 {standard sound header}
extSH = $01 {extended sound header}
cmpSH = $02 {compressed sound header}
```

The extended sample header (extSH) is the in-memory implementation of the Audio Interchange File Format standard expected by the Sound Manager. The AIFF standard specifies up to 32 bit sample sizes, up to 128 channels per file, and much more. Refer to the AIFF documentation for more details. The compressed sample header (cmpSH) is the compressed sample counter-part of the extended sample header. Refer to the Macintosh Audio Compression and Expansion documentation for further information.

Developers are free to use their own encode options with values in the range 64-127. Apple reserves the values 0 - 63.

The baseNote is the pitch at which the original sample was taken. If a harpsichord were sampled while playing middle C, then the baseNote is middle C. The `baseNote` values are 1 through 127 inclusive. (Refer to Table 4.) The `baseNote` allows the Sound Manager to calculate the proper play back rate of the sample when an application uses the `noteCmd`. Applications should not modify the `baseNote` of a sampled sound. To use the sample at different pitches, send the `noteCmd` or `freqCmd`.

Refer to the section "Current Sound Manager" regarding limitations with the `noteCmd` and `freqCmd`.

Each byte in the `sampleArea` data is similar in value to those in a wave table description. Each byte is a value of `$00` through `$FF` inclusive; `$80` represents a zero level of amplitude, `$00` is the largest negative value, and `$FF` is the largest positive value.

The Sound Manager Summary contains the description of the data format to be used with 16 bit sampled sounds. Developers wishing to write custom synthesizers for their hardware are encouraged to use this data format. This data structure is intended to complement the use of the AIFF standard.

SOUND RESOURCES

The 'snd ' Resource

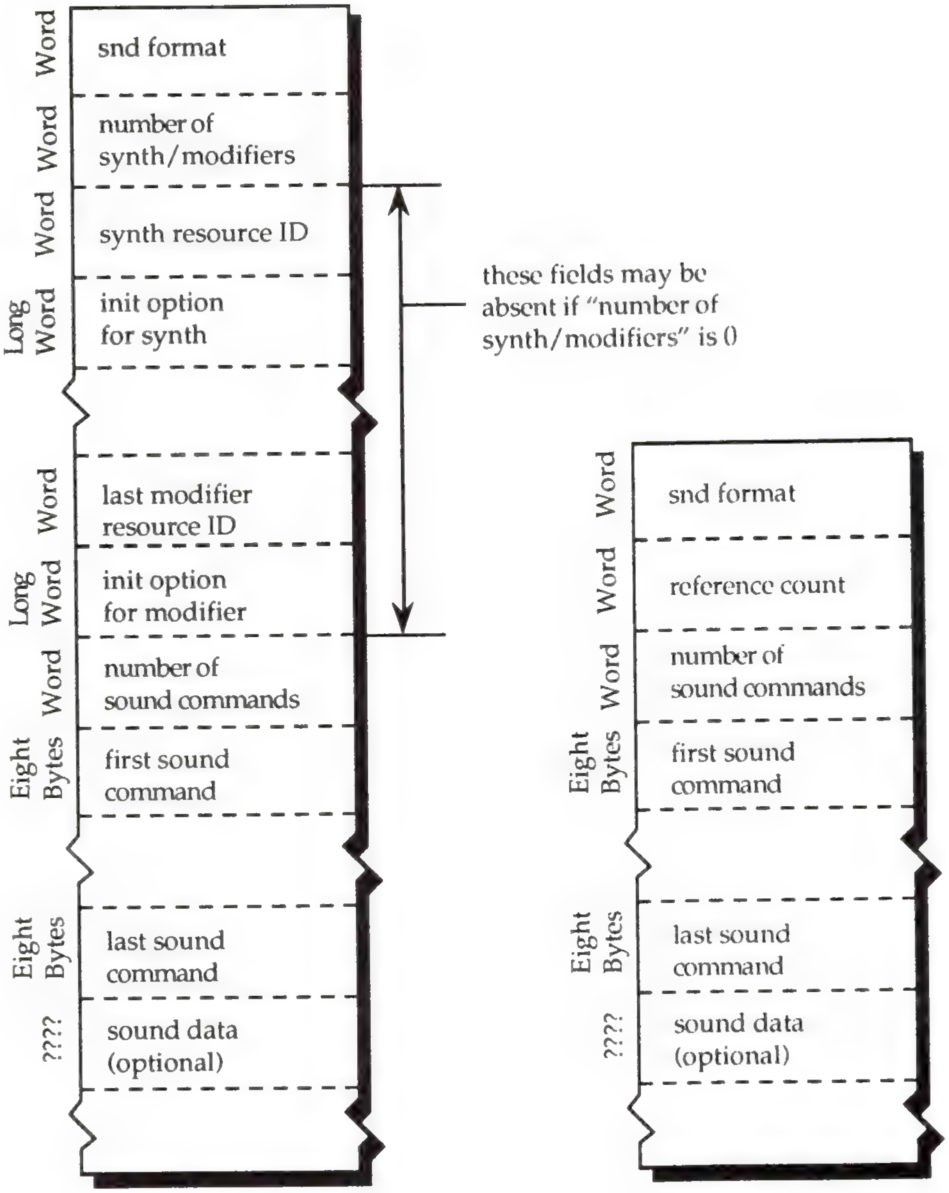


Figure 3 'snd ' Resource Layout

Sound resources are intended to be simple, portable, and dynamic solutions for incorporating sounds into applications. Creating these 'snd ' or sound resources, requires some understanding of sound synthesis to build a sampled sound header, wave table data, and sound commands. There are two types of 'snd ' resources, format 1 and format 2. Figure 3 compares the structures of both of these formats. These resources should have their purgeable bit set or the application will need to call _HPurge after using the 'snd '.

The format 1 'snd ' was developed for use with the Sound Manager. A format 1 'snd ' may be a sequence of commands describing a melody without specifying a synthesizer or modifier and without sound data. This would allow an application to use the _SndPlay routine on any channel to play that melody. A format 1 'snd ' resource may contain a sampled sound or wave table data.

The format 2 'snd ' was developed for use with HyperCard. It is intended for use with the sampled sound synthesizer only. A format 2 simply contains a sound command that points to a sampled sound header.

HyperCard (versions 1.2.1 and earlier) contain 'snd ' resources incorrectly labeled as format 1. Refer to Macintosh Technical Note #168.

Numbers for 'snd ' resources in the range 0 through 8191 are reserved for Apple. The 'snd ' resources numbered 1 through 4 are defined to be the standard system beep.

A sound command contained in a 'snd ' resource with associated sound data is marked by setting the high bit of the command. This changes the param2 field of the command to be an offset value from the resource's beginning, pointing to the location of the sound data. Refer to Figure 5 showing the structure of a sound command. To calculate this offset, use one of the following formulas below.

For a format 1 'snd ' resource, the offset is calculated as follows:

```
offset = 4 + (number of synth/mods * 6) + (number of cmds * 8)
```

For a format 2 'snd ' resource, the offset is calculated as follows:

```
offset = 6 + (number of cmds * 8)
```

The first few bytes of the resource contain 'snd ' header information and are a different size for either format. Each synthesizer or modifier specified in a format 1 'snd ' requires 6 bytes. The number of synthesizers and/or modifiers multiplied by 6 is added to this offset. The number of commands multiplied by 8 bytes, the size of a sound command, is added to the offset.

Format 1 'snd ' Resource

Figure 3 shows the fields of a format 1 'snd ' resource. This resource may also contain the actual sound data for the wave table synthesizer or the sampled sound synthesizer. The number of synthesizer and modifiers to be used by this 'snd ' is specified in the field `number of synth/modifiers`. The synthesizer required to produce the sound described in the 'snd ' is specified by the field `synth resource ID`. If any modifiers are to be installed, their resource IDs follow the first synthesizer. Any synthesizer or modifier specified beyond this first one will be installed into the channel as a modifier.

For every synthesizer and modifier, an `init` option can be supplied in the field immediately following the `resource ID` for each synthesizer or modifier. The number of commands within the resource is specified in the field `number of sound commands`. Each sound command follows in the order they should be sent to the channel. If a command such as a `bufferCmd` is contained in this resource, it needs to specify where in the resource the sampled sound header is located. This is done by setting the high bit of the `bufferCmd` and supplying the offset in `param2`. Refer to the section "Sound Manager Commands".

The 'snd ' resource may be only a sequence of commands describing a melody playable by any synthesizer. This allows the 'snd ' to be used on any channel. In this case the `number of synth/modifiers` should be 0, and there would not be a `synth resource ID` nor `init option` in the 'snd '.

Example Format 1 'snd '

The following example resource contains the proper information to create a sound with `_SndPlay` and the sampled sound synthesizer.

```
HEX         Size Meaning

{beginning of snd resource, header information}
$0001       WORD  format 1 resource
$0001       WORD  number of synth/modifiers to be installed

{synth ID to be used}
$0005       WORD  resource ID of the first synth/modifier
$0000 0000  LONG  initialization option for first synth/modifier

$0001       WORD  number of sound commands to follow

{first command, 8 bytes in length}
$8051       WORD  bufferCmd, high bit on to indicate sound data included
$0000       WORD  bufferCmd param1
$0000 0014  LONG  bufferCmd param2, offset to sound header (20 bytes)

{sampled sound header used in a soundCmd and bufferCmd}
$0000 0000  LONG  pointer to data (it follows immediately}
$0000 0BB8  LONG  number of samples in bytes (3000 samples)
$56EE 8BA3  LONG  sampling rate of this sound (22kHz)
$0000 07D0  LONG  starting of the sample's loop point
$0000 0898  LONG  ending of the sample's loop point
$00         BYTE  standard sample encoding
$3C         BYTE  baseNote (middle C) at which sample was taken

{Packed Array [1..3000] OF Byte, the sampled sound data}
$8080 8182 8487 9384 6F68 6D65 727B 8288
$918E 8D8F 867E 7C79 6F6D 7170 7079 7F81
$898F 8D8B...
```

Format 2 'snd ' Resource

The format 2 'snd ' resource is used by the sampled sound synthesizer only and must contain a sampled sound. The _SndPlay routine supports this format by automatically opening a channel to the sample sound synthesizer and using the bufferCmd.

Figure 3 shows the fields of a format 2 'snd ' resource. The field reference count is for the application's use and is not used by the Sound Manager. The fields number of sound commands and the sound commands are the same as described in a format 1 resource. The last field of this 'snd ' is for the sampled sound. The first command should be either a soundCmd or bufferCmd with the pointer bit set in the command to specify the location of this sampled sound header. Any other sound commands in this 'snd ' will be ignored by the Sound Manager.

Example Format 2 'snd '

The following example resource contains the proper information to create a sound with _SndPlay and the sampled sound synthesizer.

```
HEX         Size Meaning

{beginning of 'snd ' resource, header information}
$0002        WORD  format 2 resource
$0000        WORD  reference count for application's use
$0001        WORD  number of sound commands to follow

{first command, 8 bytes in length}
$8051        WORD  bufferCmd, high bit on to indicate sound data included
$0000        WORD  bufferCmd param1
$0000 0014   LONG  bufferCmd param2, offset to sound header (20 bytes)

{sampled sound header used in a soundCmd and bufferCmd}
$0000 0000   LONG  pointer to data (it follows immediately}
$0000 0BB8   LONG  number of samples in bytes (3000 samples)
$56EE 8BA3   LONG  sampling rate of this sound (22kHz)
$0000 07D0   LONG  starting of the sample's loop point
$0000 0898   LONG  ending of the sample's loop point
$00          BYTE  standard sample encoding
$3C          BYTE  baseNote (middle C) at which sample was taken

{Packed Array [1..3000] OF Byte, the sampled sound data}
$8080 8182 8487 9384 6F68 6D65 727B 8288
$918E 8D8F 867E 7C79 6F6D 7170 7079 7F81
$898F 8D8B...
```

The `'snth'` Resource

The `'snth'` resources are the routines that get linked to a sound channel used to create sound. The calls to `_SndPlay`, `_SndNewChannel`, `_SndAddModifier`, and `_SndControl` are mapped with unique `'snth'` resources based on the hardware present on each Macintosh. The Sound Manager first determines the type of Macintosh being used. Then, using the `id` specified in one of the four routines above, adds a constant to this `id`. For the Macintosh Plus and SE, a constant of `$1000` is added to this `id`. For the Macintosh II, `$800` is added to the `id`. If the mapped resource ID is not available, the Sound Manager will use the actual `id` value specified.

The `'snth'` resource IDs in the range 0 through 255 inclusive are reserved for Apple within the `'snth'` resource mapping range.

Resource ID	Synthesizer	Target Macintosh
$0001	noteSynth	general for any Macintosh
$0003	waveTableSynth	general for any Macintosh
$0005	sampledSynth	general for any Macintosh
$0006-$00FF	*reserved for Apple*	general for any Macintosh
$0100-$0799	*free for developers*	general for any Macintosh
$0801	noteSynth	Mac with Apple Sound Chip
$0803	waveTableSynth	Mac with Apple Sound Chip
$0805	sampledSynth	Mac with Apple Sound Chip
$0806-$08FF	*reserved for Apple*	Mac with Apple Sound Chip
$0900-$0999	*free for developers*	Mac with Apple Sound Chip
$1001	noteSynth	Mac Plus and SE
$1003	waveTableSynth	Mac Plus and SE
$1005	sampledSynth	Mac Plus and SE
$1006-$10FF	*reserved for Apple*	Mac Plus and SE
$1100-$1199	*free for developers*	Mac Plus and SE

Table 2 Synthesizer Resource IDs

For example, if an application requested the sampled sound synthesizer while running on the Macintosh Plus, it uses the resource ID of 5 when calling `_SndNewChannel`. The Sound Manager will then open the `'snth'` resource with the ID of `$1005` since this synthesizer is specific to the Macintosh Plus. Table 2 lists the current synthesizers and the IDs used by each Macintosh.

Refer to the section "Current Sound Manager" regarding the Macintosh II `'snth'` IDs.

SOUND MANAGER ROUTINES

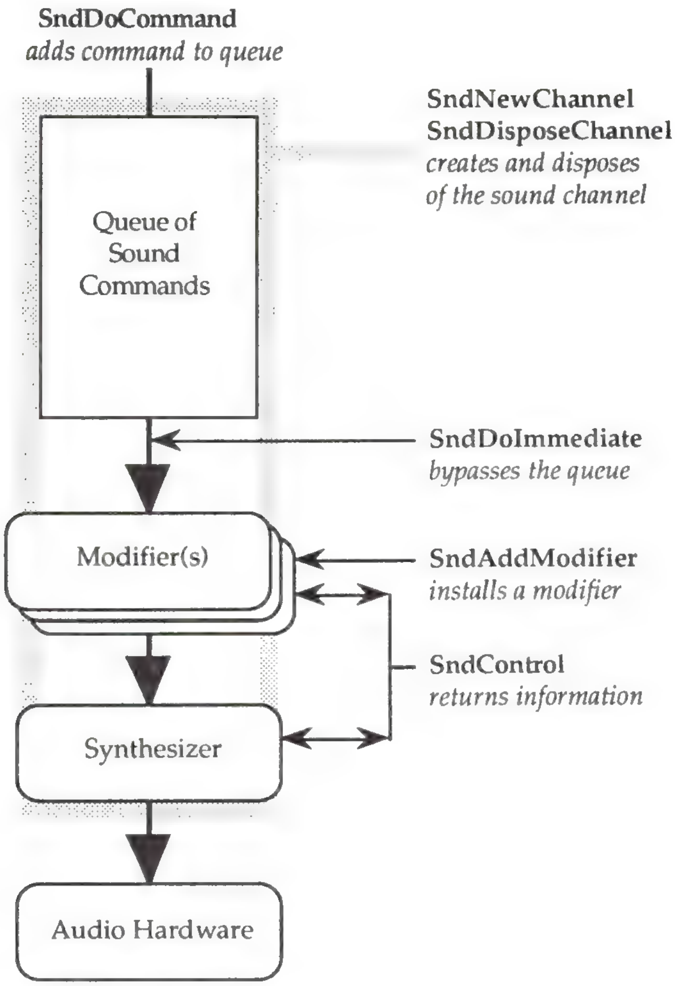


Figure 4 Sound Channel and Routines

```
FUNCTION SndPlay (chan: SndChannelPtr; sndHdl: Handle; async: BOOLEAN)
                    : OSErr;
```

The function `_SndPlay` is a higher level sound routine and is generally used separately from the other Sound Manager calls. `_SndPlay` will attempt to play the sound specified in the `'snd '` resource located at `sndHdl`. This is the only Sound Manager routine that accepts a `'snd '` resource as one of its parameters. If a format 1 `'snd '` specifies a synthesizer and any modifiers, those `'snth'` resource(s) will be loaded in memory and linked to the channel. All commands contained in the `'snd '` will be sent to the channel. If the application passes `NIL` as the channel pointer, `_SndPlay` will create a channel in the application's heap. The Sound Manager will release this memory after the sound has completed. The async parameter is ignored if NIL is passed as the channel pointer.

If the application does supply a channel pointer in chan, the sound can be produced asynchronously. When sound is played asynchronously, a completion routine can be called when the last command has finished processing. This procedure is the userRoutine supplied with _SndNewChannel. _SndPlay will call _HGetState on the 'snd ' resource before _HMoveHi and _HLock, and once the sound has completed, will restore the state of the 'snd ' resource's handle with _HSetState.

If the format 1 'snd ' resource does not specify which synthesizer is to be used, _SndPlay will default to the note synthesizer. _SndPlay will also support a format 2 'snd ' resource using the sampled sound synthesizer and a bufferCmd. Note that a format 1 'snd ' must use have a bufferCmd in order to be used with _SndPlay and the sampled sound synthesizer.

⚠ Do not use _SndPlay with a 'snd ' that specifies a synthesizer ID if the channel has already been linked to a synthesizer.

```
FUNCTION SndNewChannel (VAR chan: SndChannelPtr; synth: INTEGER;
                        init: LONGINT; userRoutine: ProcPtr) : OSErr;
```

When NIL is passed as the chan parameter, _SndNewChannel will allocate a sound channel record in the application's heap and return its POINTER. Applications concerned with memory management can allocate their own channel memory and pass this POINTER in the chan parameter. Typically this should not present a problem since a channel should only be in use temporarily. Each channel will hold 128 commands as a default size. The length of a channel can be expanded by the application creating its own channel in memory.

The synth parameter is used to specify which synthesizer is to be used. The application specifies a synthesizer by its resource ID, and this 'snth' resource will be loaded and linked to the channel. The state of the 'snth' handle will be saved with _HGetState. To create a channel without linking it with a synthesizer, pass 0 as the synth. This is useful when using _SndPlay with a 'snd ' that specifies a synthesizer ID.

The application may specify an init option that should be sent to the synthesizer when opening the channel. For example, to open the third wave table channel use initChan2 as the init. Only the wave table synthesizer and sampled sound synthesizer currently use the init options. To determine if a particular option is available by the synthesizer, use the availableCmd.

```
initChanLeft      = $02; {left channel - sampleSynth only}
initChanRight     = $03; {right channel- sampleSynth only}
initChan0         = $04; {channel 1 - wave table only}
initChan1         = $05; {channel 2 - wave table only}
initChan2         = $06; {channel 3 - wave table only}
initChan3         = $07; {channel 4 - wave table only}
initSRate22k      = $20; {22k sampling rate - sampleSynth only}
initSRate44k      = $30; {44k sampling rate - sampleSynth only}
initMono          = $80; {monophonic channel - sampleSynth only}
initStereo        = $C0; {stereo channel - sampleSynth only}
```

Refer to the section "Current Sound Manager" regarding init options and the sampled sound synthesizer.

If an application is to produce sounds asynchronously or needs to be alerted when a command has completed, it uses a CallBack procedure. This routine will be called once the callBackCmd has been received by the synthesizer. If you pass NIL as the userRoutine, then any callBack command will be ignored.

```
FUNCTION SndAddModifier (chan: SndChannelPtr; modifier: ProcPtr;
                         id: INTEGER; init: LONGINT) : OSErr;
```

This routine is used to install a modifier into an open channel specified in chan. The modifier will be installed in front of the synthesizer or any existing modifiers in the channel. If the modifier is saved as a 'snth' resource, pass NIL for the ProcPtr and specify its resource ID in the parameter id. This will cause the Sound Manager to load the 'snth' resource, lock it in memory, and link it to the channel specified. The state of the 'snth' resource handle will be saved with _HGetState. Refer to the section "User Routines" for more information regarding writing a modifier.

Refer to the section "Current Sound Manager" regarding modifier resources.

```
FUNCTION SndDoCommand (chan: SndChannelPtr; cmd: SndCommand;
                        noWait: BOOLEAN) : OSErr;
```

This routine will send the sound command specified in cmd to the existing channel's command queue. If the parameter noWait is set to FALSE and the queue is full, the Sound Manager will wait until there is space to add the command. If noWait is set to TRUE and the channel is full, the Sound Manager will not send the command and returns the error "queueFull".

```
FUNCTION SndDoImmediate (chan: SndChannelPtr; cmd: SndCommand): OSErr;
```

This routine will bypass the command queue of the existing channel and send the specified command directly to the synthesizer, or the first modifier. This routine will also override any waitCmd, pauseCmd or syncCmd that may have been received by the synthesizer or modifiers.

```
FUNCTION SndControl (id: INTEGER; VAR cmd: SndCommand) : OSErr;
```

This routine is used to send control commands directly to a synthesizer or modifier specified by its resource ID. This can be called even if no channel has been created for the synthesizer. This control call is used with the availableCmd or versionCmd to request information regarding a synthesizer. The result of this call is returned in cmd.

```
FUNCTION SndDisposeChannel (chan: SndChannelPtr; quietNow: BOOLEAN) :
                             OSErr;
```

This routine will dispose of the channel specified in chan and release all memory created by the Sound Manager. If an application created its own channel record in memory or installed a sound as an instrument, the Sound Manager will not dispose of that memory. The Sound Manager will restore the original state of 'snth' resource handles with a call to _HSetState.

_SndDisposeChannel can either immediately dispose of a channel or wait until the queued commands are processed. If quietNow is set to TRUE, a flushCmd and then a quietCmd is sent to the channel. This will remove all commands, stop any sound in progress and close the channel. If quietNow is set to FALSE, then the Sound Manager will issue a quietCmd only and wait until the quietCmd is received by the synthesizer before disposing of the channel. In this situation _SndDisposeChannel will be synchronous.

SOUND MANAGER COMMANDS

Command Descriptions

Sound commands are placed into a channel one after the other. At the end of the channel is the synthesizer which interprets the command and plays the sound with the hardware. All synthesizers are designed to accept the most general set of sound commands. Some commands are specific to only a particular synthesizer. There are some commands and options that may not be currently implemented by a synthesizer. Refer to section "The Current Sound Manager" for more details.

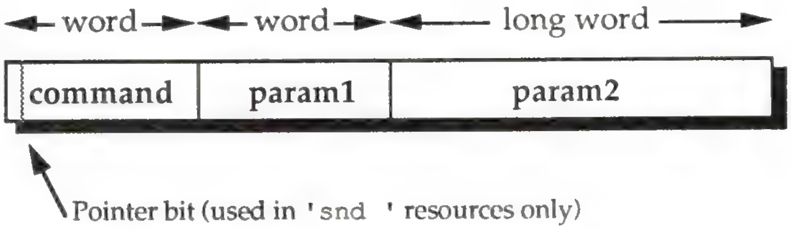


Figure 5 Generic Command Format

Figure 5 shows the structure of a generic sound command. Commands are always eight bytes in length. The first two bytes are the command number, and the next six make up the command's options. The format of these last six bytes will depend on the command being used.

The `pointer bit` is only used by `'snd '` resources that contain commands and associated sound data (i.e. sampled sound or wave table data). If the high bit of the command is set, then `param2` is an offset specifying where the associated data is located. This offset is the number of bytes starting from the beginning of the resource to the associated sound data. The section "Sound Resources" shows how this offset is calculated.

`cmd=nullCmd`   `param1=0`   `param2=0`

This command is sent by modifiers. It is simply absorbed by the Sound Manager and no action is performed. Modifiers use a `nullCmd` to replace commands in a channel to prevent them from being sent to a synthesizer.

```
cmd=initCmd         param1=0            param2=init
```

This command is only sent by the Sound Manager. It will send an `initCmd` to the synthesizer when an application uses the routines `_SndPlay`, `_SndNewChannel` or `_SndAddModifier`. This causes a synthesizer or modifier to allocate its private memory storage and to use the `init` option.

```
cmd=freeCmd         param1=0            param2=0
```

This command is only sent by the Sound Manager. It is exactly opposite of the `initCmd`. When an application calls `_SndDisposeChannel`, the Sound Manager will send the `freeCmd` to the synthesizer. This causes the synthesizer to dispose of all the private memory it had allocated.

```
cmd=quietCmd        param1=0            param2=0
```

This command is sent by an application using `_SndDoImmediate`. It will cause the synthesizer to stop any sound in progress. It is also sent by the Sound Manager with the `_SndDisposeChannel` routine.

```
cmd=flushCmd        param1=0            param2=0
```

This command is sent by an application using `_SndDoImmediate`. It will cause all commands in the channel be be removed. It is also sent by the Sound Manager from `_SndDisposeChannel` when `quietNow` is `TRUE`.

```
cmd=waitCmd         param1=duration     param2=0
```

This command is sent by an application or a modifier. It will suspend all processing in the channel for the number of half-milliseconds specified in `duration`. A one second wait would be a `duration` of `2000`.

```
cmd=pauseCmd        param1=0            param2=0
```

This command is sent by an application or a modifier to cause the channel to suspend processing until a `tickleCmd` or `resumeCmd` is received.

```
cmd=resumeCmd       param1=0            param2=0
```

This command is sent by an application or a modifier to cause a channel to resume processing of commands. This is the opposite of the `pauseCmd`.

```
cmd=callBackCmd     param1=user-defined
                    param2=user-defined
```

This command is sent by an application. The callBackCmd causes the Sound Manager to call the userRoutine specified in _SndNewChannel. The two parameters of this command can be used by the application for any purpose. This allows an application to have a general userRoutine for any channel. By using param1 and param2 with unique values, the CallBack procedure can test for specific actions to take. Refer to the section "User Routines".

This command is used as a marker for an application to determine at what point the channel has reached in processing its queue. It is mostly used to determine when to dispose of a channel, since the callBackCmd is generally the last command sent. It can also be used to allow an application to synchronize sounds with other actions.

```
cmd=syncCmd         param1=count        param2=identifier
```

This command is sent by an application. Every syncCmd is held in the channel, suspending any further processing until its count equals 0. The Sound Manager will first decrement the count and then wait for another syncCmd having the same identifier to be received on another channel.

To synchronize four wave table channels, send the syncCmd to each channel with count = 4 giving each command the same identifier. If a channel should wait for two more syncCmds, then its count would be 3. If a channel is to wait for one more syncCmd, its count would be sent as 2.

Refer to the section "Current Sound Manager" regarding the count parameter of a syncCmd.

```
cmd=emptyCmd        param1=0            param2=0
```

This command is only sent by the Sound Manager. Synthesizers expect to receive additional commands after a resumeCmd. If no other commands are to be sent, the Sound Manager will send an emptyCmd.

```
cmd=tickleCmd       param1=0            param2=0
```

This command is only sent by the Sound Manager to a modifier. This will cause modifiers to perform their requested periodic actions. If the tickleCmd had been requested by a howOftenCmd, then a tickleCmd will be sent periodically according to the period specified in the howOftenCmd. If the tickleCmd had been requested by an wakeUpCmd, then this command will be sent only once according to the period specified in the wakeUpCmd. A tickleCmd command will also resume a channel suspended by a pauseCmd.

cmd=requestNextCmd   param1=count param2=0

This command is only sent by the Sound Manager in response to a modifier returning TRUE. Refer to the section "User Routine" discussing modifiers. Count is the number of consecutive times that the modifier has requested another command.

cmd=howOftenCmd   param1=period   param2=pointer

This command is sent by a modifier and will instruct the Sound Manager to periodically send a tickleCmd. Param1 contains the period (in half-milliseconds) that a tickleCmd should be sent. Param2 contains a POINTER to the modifier stub.

cmd=wakeUpCmd   param1=period   param2=pointer

This command is sent by a modifier and will instruct the Sound Manager to send a single tickleCmd after the period specified (in half-milliseconds). Param2 contains a POINTER to the modifier stub.

The howOftenCmd and the wakeUpCmd are mutually exclusive. Sending one will cancel the other.

cmd=availableCmd param1=result   param2=init

This command is sent by an application to determine if certain characteristics specified in the init parameter are available from the synthesizer. This command can only be used with the _SndControl routine. These init options are documented under the _SndNewChannel routine and are passed in param2 of the availableCmd.

```
myCmd.cmd := availableCmd;
myCmd.param1 := 0;
myCmd.param2 := initStereo;   {we'll test for a stereo channel}
myErr := SndControl (sampledSynth, myCmd);
IF (myCmd.param1 <> 0) THEN stereoAvailable := TRUE;
```

The result is returned in param1. A result of 1 is returned if the synthesizer has the requested characteristics. If it does not, the result is 0.

Refer to section "Current Sound Manager" regarding limitations with the availableCmd.

cmd=versionCmd param1=0 param2=version

This command is sent by applications and the Sound Manager to determine which version of the synthesizer is available. The `versionCmd` can only be sent with the `_SndControl` routine. The `version` is returned in `param2`. Version 1.2 of a synthesizer would be returned as `$0001 0002`.

cmd=noteCmd param1=duration
param2=amplitude + frequency

This command is sent by applications and modifiers to specify a note for either the note synthesizer, or with an instrument installed into the channel. The `duration` parameter is in half-milliseconds. A `duration` of `2000` would be a duration of one second. The maximum `duration` is a `duration` of `32767` or about 16 seconds. The structure of a `noteCmd` is given in Figure 6.

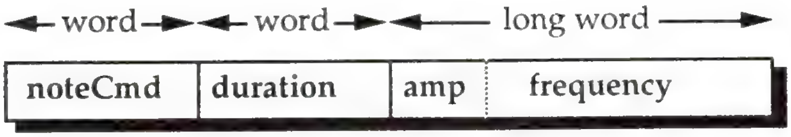


Figure 6 noteCmd Format

The `param2` of a `noteCmd` is a combination of an `amplitude` and a `frequency`. The `amplitude` is passed in the high byte and the lower three bytes are the `frequency` . The `frequency` can be specified in two ways, as a decimal note (refer to the section "Note Values and Durations") or a frequency value (refer to `freqCmd`). The `amplitude` values range from `$00` to `$FF` inclusively. The following example demonstrates the use of a `noteCmd`.

```
amp := $FF000000;               {loudest possible amplitude}
note := 60;                     {middle C}
myCmd.cmd := noteCmd;
myCmd.param1 := 2000;           {one second duration}
myCmd.param2 := amp + note;
myErr := SndDoCommand(myChan, myCmd, FALSE);
```

The `noteCmd` will start at the beginning of a sampled sound. The `noteCmd` uses the loop points of the header to extend the length of the sound to the `duration` specified in a `noteCmd`. There must be a loop ending point specified in the header in order for the `noteCmd` to work properly.

Refer to the section "Current Sound Manager" regarding limitations with the `noteCmd` and using `amplitude`.

```
cmd=restCmd          param1=duration    param2=0
```

This command is sent by applications and modifiers to cause the channel to rest for the `duration` specified in half-milliseconds.

```
cmd=freqCmd          param1=0           param2=frequency
```

This command is sent by applications and modifiers. A `frequency` can be sent to a synthesizer to change the pitch of a sound. It is similar to the `noteCmd` in that a decimal note value can be used instead of a frequency value. The structure of this command is shown in Figure 7. If no sound is playing, it causes the synthesizer to begin playing at the specified `frequency` for an indefinite duration. The upper byte of `param2` is ignored. A frequency value is sent in the lower three bytes of `param2`, where the frequency desired is multiplied by 256. For example, to specify a frequency of 440 Hz (the A below middle C) the `frequency` value would be `440 * 256` or `112640`.

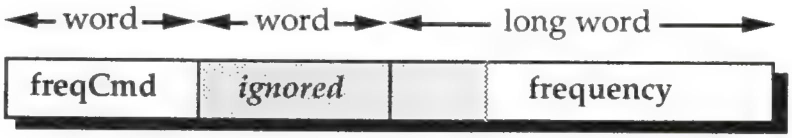


Figure 7 freqCmd format

Refer to the section "Current Sound Manager" regarding the limitations of the `freqCmd`.

```
cmd=ampCmd           param1=amplitude   param2=0
```

This command is sent by applications and modifiers to change the `amplitude` of the sound in progress. If no sound is currently playing, then it will affect the `amplitude` of the next sound.

Refer to the section "Current Sound Manager" regarding the use of `amplitude`.

```
cmd=timbreCmd        param1=timbre      param2=0
```

This command is sent by applications and modifiers. It is used only by the note synthesizer to change its timbre or tone. A sine wave is specified as `0` in param1 and produces a flute-like sound. A value of 255 in param1 represents a modified square wave and produces a buzzing or reed-like sound. Changing the note synthesizer's timbre should be done before playing the sound. Only a Macintosh with the Apple Sound Chip will allow this command to be sent while a sound is in progress.

cmd=waveTableCmd   param1=length   param2=pointer

This command is sent by applications. It is only used by the wave table synthesizer. It will install a wave table to be used as an instrument by supplying a POINTER to the wave table in param2.

All wave cycles will be re-sampled to 512 bytes.

cmd=phaseCmd   param1=shift   param2=pointer

This command is sent by applications. It is only used by the wave table synthesizer to synchronize the phases of the wave cycles across different wave table channels. As an example, if two wave table channels containing the same wave cycle were sent the same noteCmd, they could not begin exactly at the same time. Therefore, to synchronize the wave cycles for these two channels the phaseCmd is sent.

This prevents the phasing effects of playing two similar waves together at the same pitch. The channel will have its wave shifted by the amount specified in shift to correspond with the wave's phase in the channel specified in param2. The shift value is a 16 bit fraction going from zero to one. The value of $8000 would be the half-way point of the wave cycle. Generally, the effects from this command will not be noticed.

Refer to the section "Current Sound Manager" regarding the phaseCmd.

cmd=soundCmd   param1=0   param2=pointer

This command is sent by an application and is only used by the sampled sound synthesizer. If the application sends this command, param2 is a POINTER to the sampled sound locked in memory. The format of a sampled sound is shown in section "The Sampled Sound Synthesizer". This command will install the sampled sound as an instrument for the channel. If the soundCmd is contained within a 'snd ' resource, the high bit of the command must be set. To use a sampled sound 'snd ' as an instrument , first obtain a POINTER to the sampled sound header locked in memory. Then pass this POINTER in param2 of a soundCmd. After using the sound, the application is expected to unlock this resource and allow it to be purged.

cmd=bufferCmd param1=0 param2=pointer

This command is sent by applications and the Sound Manager to play a sampled sound, in one-shot mode, without any looping. The POINTER in param2 is the location of a sampled sound header locked in memory. The format of a sampled sound is shown in section "The Sampled Sound Synthesizer". A bufferCmd will be queued in the channel until the preceding commands have been processed. If the bufferCmd is contained within a 'snd ' resource, the high bit of the command must be set. If the sound was loaded in from a 'snd ' resource, the application is expected to unlock this resource and allow it to be purged after using it.

Refer to the section "Current Sound Manager" regarding the bufferCmd.

cmd=rateCmd param1=0 param2=rate

This command is sent by applications to modify the pitch of the sampled sound currently playing. The current pitch is multiplied by the rate in param2. It is used for pitch bending effects. The default rate of a channel is 1.0. To cause the pitch to fall an octave (or half of its frequency), send the rateCmd with param2 equal to one half as shown below.

```
myCmd.cmd := rateCmd;
myCmd.param1 := 0;
myCmd.param2 := FixedRatio(1, 2);
myErr := SndDoImmediate(myChan, myCmd);
```

cmd=continueCmd param1=0 param2=pointer

This command is sent by applications to the sampled sound synthesizer. It is similar to the bufferCmd. Long sampled sounds may be broken up into smaller sections. In this case, the application would use the bufferCmd for the first portion and the continueCmd for any remaining portions. This will result in a single continuous sound with the first byte of the sample being joined with the last byte of the previous sound header without audible clicks.

Refer to the section "Current Sound Manager" regarding the continueCmd.

USER ROUTINES

These user routines will be called at interrupt time and therefore must not attempt to allocate, move or dispose of memory, de-reference an unlocked handle, or call other routines that do so. Assembly language programmers must preserve all registers other than `A0-A1`, and `D0-D2`. If these routines are to use an application's global data storage, it must first reset `A5` to the application's `A5` and then restore it upon exit. Refer to Macintosh Technical Note #208 regarding setting up `A5`.

```
PROCEDURE CallBack(chan: SndChannelPtr; cmd: SndCommand);
```

The function `_SndNewChannel` allows a completion routine or `CallBack` procedure to be associated with a channel. This procedure will be called when a `callBackCmd` is received by the synthesizer linked to that channel. This procedure can be used for various purposes. Generally it is used by an application to determine that the channel has completed its commands and to dispose of the channel. The `CallBack` procedure itself cannot be used to dispose of the channel, since it may be called at interrupt time.

A `CallBack` procedure can also be used to signal that a channel has reached a certain point in the queue. An application may wish to perform particular actions based on how far along the sequence of commands a channel has processed. Applications can use `param1` or `param2` of the `callBackCmd` as flags. Based on certain flags for certain channels, the call back can perform many different functions. The `CallBack` procedure will be passed the channel that received the `callBackCmd`. The entire `callBack` command is also passed to the `CallBack` procedure.

```
myCmd.cmd := callBackCmd;      {install the callBack command}
myCmd.param1 := 0;             {not used in this example}
myCmd.param2 := SetCurrentA5; {pass the callBack our A5}
myErr := SndDoCommand (myChan, myCmd, FALSE);
```

The example code above is used to setup a `callBackCmd`. Note that `param2` of a sound command is a `LONGINT`. This can be used to pass in the application's `A5` to the `CallBack` procedure. Once this command is received by the synthesizer, the following example `CallBack` procedure can set `A5` in order to access the application's globals. The function's `SetCurrentA5` and `SetA5` are documented in Macintosh Technical Note #208.

```
Procedure SampleCallBack (theChan: SndChannelPtr; theCmd: SndCommand);

VAR
   theA5 : LONGINT;

BEGIN
   theA5 := SetA5(myCmd.param2);    {set A5 and get current A5}
   callBackPerformed := TRUE;       {global flag}
   theA5 := SetA5(theA5);           {restore the current A5}
END;
```

```
FUNCTION Modifier(chan: SndChannelPtr; VAR cmd: SndCommand;
                        mod: ModifierStubPtr) : BOOLEAN
```

A modifier will be called when the command reaches the end of the queue, before being sent to the synthesizer or other modifiers that may be installed. Chan will contain the channel pointer allowing multiple wave table channels to be supported by the same modifier. The ModifierStub is a record created by the Sound Manager during the call _SndAddModifier. A pointer to the ModifierStub is in mod. There are two special commands that the modifier must support, the initCmd and the freeCmd.

Refer to the section "Current Sound Manager" regarding modifiers being saved as resources.

```
ModifierStub = PACKED RECORD
               nextStub:      ModifierStubPtr; {pointer to next stub}
               code:          ProcPtr;    {pointer to modifier}
               userInfo:      LONGINT;    {free for modifier's use}
               count:         Time;       {used internally}
               every:         Time;       {used internally}
               flags:         SignedByte; {used internally}
               hState:        SignedByte; {used internally}
            END;
```

The initCmd is sent by the Sound Manager when an application calls _SndAddModifier. This is a command telling the modifier to allocate any additional data. The ModiferStub contains a four byte field, userInfo, that can be used as a pointer to this additional memory. The initCmd will not be sent to a modifier at interrupt time. This allows a modifier to allocate memory and save the current application's A5. All memory storage allocated by the modifier must be locked, since the modifier will be called at interrupt time.

The freeCmd will be sent to the modifier when the Sound Manager is disposing of the channel. This command will not be sent at interrupt time. At this point the modifier should free any data it may have allocated.

A modifier will be given the current command, before the command is sent to the synthesizer or other modifiers. The current command is sent to the modifier in the variable cmd. A nullCmd is never sent to a modifier. If the modifier wished to ignore the current command and allow it to be sent on, it would return FALSE. To remove the current command, replace it with a nullCmd and then return FALSE. To alter the current command, replace it with the new one and return FALSE. Returning FALSE means that the modifier has completed its function.

If the modifier is to send additional commands to the channel, the function will return TRUE and may or may not change the current command. The Sound Manager will call the modifier again sending it a requestNextCmd. The modifier can then replace this command with the one desired. The modifier can continue to return TRUE to send additional commands. The requestNextCmd will indicate the number of times this command has been consecutively sent to the modifier.

⚠ Modifiers are short routines used to perform real-time modifications on channels. Having too many modifiers, or a lengthy one, may degrade performance.

THE CURRENT SOUND MANAGER

Synthesizer Details

This section documents the details for each of the current synthesizers.

The Note Synthesizer

- The version shipped with System 6.0.2 is `$0001 0002`.

- Commands currently supported:

```
availableCmd      versionCmd       freqCmd
     noteCmd         restCmd      flushCmd
    quietCmd          ampCmd     timbreCmd
```

Limitations of the Note Synthesizer

- Amplitude change is only supported by a Macintosh with the Apple Sound Chip, and is not supported by a Macintosh Plus or Macintosh SE.

- Only a single monophonic channel can be used.

The Wave Table Synthesizer

- The version shipped with System 6.0.2 is `$0001 0002`.

- Commands currently supported:

```
availableCmd      versionCmd       freqCmd
     noteCmd         restCmd      flushCmd
    quietCmd    waveTableCmd
```

Limitations of the Wave Table Synthesizer

- This synthesizer is not functioning on a Macintosh Plus or Macintosh SE.

- A maximum of four channels can be open at any time.

- Amplitude change is not supported on any Macintosh.

- The one-shot mode is not supported on any Macintosh.

- The `phaseCmd` is not working.

The Sampled Sound Synthesizer

- The version shipped with System 6.0.2 is `$0001 0002`.

- Commands currently supported:

```
availableCmd       versionCmd        freqCmd
     noteCmd          restCmd       flushCmd
    quietCmd          rateCmd       soundCmd
   bufferCmd
```

Limitations of the Sampled Sound Synthesizer

- Amplitude change is not supported on any Macintosh.

- The current hardware will only support sampling rates up to 22kHz. This is not a limitation to the playback rates, and samples can be pitched higher on playback.

- There can only be a single monophonic channel; stereo is not supported.

- The `continueCmd` is not working.

The MIDI Synthesizer

- The version shipped with System 6.0.2 is `$0001 0002`.

Limitations of the MIDI Synthesizer

- The `midiDataCmd` documented in *Inside Macintosh Volume V* cannot be used.

- Fully functional MIDI applications cannot be written using the current Sound Manager and were intended as a "poor man's" method of sending notes to a MIDI keyboard.

- A bug in the MIDI synthesizer code prevents it from working after calling `_SndDisposeChannel`.

Sound Manager Bugs

This is a list of all known bugs and possible work-arounds in the System 6.0.2 Sound Manager. Each of these issues are being addressed and are expected to be solved with the next Sound Manager release.

Macintosh II `'snth'` IDs

The System 6.0.2 `'snth'` resources for the Macintosh II are incorrectly numbered. They should be `$0801-$0805`, but were shipped as `$0001-$0005`. This does not currently present a problem for applications, since the Sound Manager will default to these versions while running on the Macintosh II.

`availableCmd`

The `availableCmd` is returning a value of `1`, meaning `TRUE`, even if the synthesizer is actually no longer available. For example, after calling `_SndNewChannel` for the `noteSynth`, the `availableCmd` for the `noteSynth` should return `FALSE` since there isn't a second one. Furthermore, considering that only one synthesizer can be active at one time, after opening the `noteSynth` the `sampledSynth` is not available, but this command reports that it is. The only time the `availableCmd` will return `FALSE` is by requesting an `init` option that a synthesizer doesn't support, such as stereo channels.

`_SndAddModifier`

A modifier resource used in multiple channels must be pre-loaded and locked in memory by the application. There is a bug when the Sound Manager is disposing of a channel causing the modifier to be unlocked, regardless of other channels that may be using that modifier. If the application locks the modifier before installing it in the channel, the Sound Manager will not unlock it, but restores its state with `_HSetState`.

`syncCmd`

This command has a bug causing the `count` to be decremented incorrectly. To synchronize four channels, the same `count = 4` should be sent to all channels. The bug is with the Sound Manager decrementing all of the `count` values with every new `syncCmd`. In order to work around this, an application can synchronize four wave table channels by sending the `syncCmd` with `count = 4`. Then a `syncCmd` with the same `identifier` is sent to the second channel, this time with `count = 3`. The third channel is sent a `syncCmd` with `count = 2`. Finally, the last channel is sent with the `count = 1`. As soon as the fourth syncCmd is received, all channels will have their count at 0 and will resume processing their queued commands. This bug will be fixed eventually, so test for the version of the synthesizer being used before relying on this.

bufferCmd

Sending a bufferCmd will reset the channel's amplitude and rate settings. Since the amplitude is already being ignored and the rate isn't typically used, this problem is not of much concern at this time.

noteCmd

This command may cause the sampled sound synthesizer to loop until another command is sent to the channel. This occurs when using a sampled sound installed as an instrument. If a noteCmd is the last command in the channel, the sound will loop endlessly. The work-around is to send a command after the final noteCmd. A callBackCmd, restCmd or quietCmd would be good.

noteCmd and freqCmd

These commands currently only support note values 1 through 127 inclusive. Refer to Table 4 for these values.

_SysBeep

On a Macintosh Plus or SE (which do not have the Apple Sound Chip) the Sound Manager will purge the application's channel of its 'snth' or sound data. The application would have to dispose of the channel at this point and recreate a new one. This is another reason to release channels as soon as the application has completed its sound. This bug can be avoided by selecting the "Simple Beep" in the Control Panel's sound 'cdev'. Applications should dispose of all channels before allowing a _SysBeep to occur. This includes putting up an alert or modal dialog that could cause the system beep. Since a foreground application under MultiFinder could cause a _SysBeep while the sound application is in the background, all applications should dispose of channels at a suspend event.

SOUND MANAGER ABUSE

Sound channels are for temporary use, and should only be created just before playing sound. Once the sound is completed, the channel should be disposed. Applications should not hold on to these channels for extended periods. The amount of overhead in `_SndNewChannel` is minimal. Basically, it is only a Memory Manager call. As long as the application holds onto a channel linked to a synthesizer, the `_SysBeep` call will not work and may cause trouble for the application's channel.

Friendly applications will dispose of all open channels during a suspend event from MultiFinder. If an application created a channel and then gets sent into the background, any foreground application or `_SysBeep` will be unable to gain access to the sound hardware.

Applications must dispose of all channels before calling `_ExitToShell`. Currently, calling `_ExitToShell` while generating a sound on the Macintosh Plus and SE will cause a system crash. So, calling `_SndDisposeChannel` before `_ExitToShell` will solve this issue. Setting `quietNow` to be `FALSE` will allow the application to complete the sound before continuing.

Do not mix older Sound Driver calls with the newer Sound Manager routines. The older Sound Driver should no longer be used. The Sound Manager is its replacement, providing all of it predecessor's abilities and more. Note that `_GetSoundVol` and `_SetSoundVol` are not part of the Sound Manager. They are used for setting parameter RAM, not the amplitude of a channel. Support for the older Sound Driver may eventually be discontinued.

The `'snd '` resource is so flexible that a warning of resource usage is needed. Most of the problems developers have with the Sound Manager are related more to the `'snd '` being used and less to the actual routines. Editing and creating `'snd '` resources with ResEdit is difficult. Many of the issues required in dealing with a `'snd '` are not supported by third party utilities. It is best to limit the `'snd '` to contain either sound data (i.e. sample sound) or a sequence of sound commands. Do not attempt to create resources that contain multiple sets of sound data.

Be very careful with what `'snd '` resources the application is intending to support. Test for the proper format and proper fields beforehand. An application needs to know the exact contents of the entire `'snd '` in order to properly handle it. Things can get ugly real quick considering variant records, variable record lengths, and the pointer math that will be required.

If an application wants to use `_SndPlay` with an existing channel already linked to a synthesizer, the `'snd '` must not contain any `synth` information. With a format 1 `'snd '`, the `number of synth/modifiers` field must be `0`, and no `synth ID` or `init option` should be in the resource. Applications can only call `_SndPlay` with a channel linked to a synthesizer using a format 1 `'snd '` that contains sound commands without `synth` information.

A format 2 `'snd '` can never be used with `_SndPlay` more than once with an existing channel. This `'snd '` is assumed to be for the sampled sound synthesizer and `_SndPlay` will link this synthesizer to the channel. If a channel is created before calling `_SndPlay` with a format 2, specify `synth = 0` in the call to `_SndNewChannel`. After calling `_SndPlay` once, the application will have to dispose of the channel before using a format 2 `'snd '` again.

FREQUENTLY ASKED QUESTIONS

Q: Is there a way to determine if a sound is being made?

A: It is not possible at this time to determine if a synthesizer is currently active or producing a sound. However, an application can use the `callBackCmd` to determine when a sound has completed.

Q: How do I determine if the Apple Sound Chip is present?

A: There is no supported method for determining this. A new `_SysEnvirons` record is being considered to contain this information.

Q: How can I use the Sound Manager for a metronome effect?

A: Use a modifier to send a `noteCmd` to the note synthesizer. The modifier will use the `howOftenCmd` to cause the Sound Manager to send a `tickleCmd`. Every time the modifier gets called, it can send a `noteCmd` to cause the click.

Q: What is the maximum number of synthesizers that can be opened at once? Can I have the `noteSynth` and the `sampledSynth` open at the same time and produce sound from either?

A: Only one synthesizer can be active at any time. This is because the active synthesizer "owns" the sound hardware until the channel is disposed of.

Q: How can I tell if more than four wave table channels are open or if another application has already open a synthesizer?

A: It is not possible at this time to determine when more than the maximum number of wave table channels has been allocated due to a limitation with the `availableCmd`. This issue is being investigated. It is not possible to determine if a synthesizer is in use by another application. If all applications would dispose of their channels at the resume event, this would not be a problem.

Q: How do I get `_SndPlay` to play the sound asynchronously? The Sound Manager seems to ignore the `async` parameter.

A: If `NIL` is used for the channel, then `_SndPlay` does ignore the `async` flag. To play the sound asynchronously, create a new channel with `_SndNewChannel` and pass this channel's pointer to `_SndPlay`. Again, if this 'snd ' contains `'snth'` information you must not link a synthesizer to the channel. Pass `0` as the `synth` in the call to `_SndNewChannel`.

Q: Should we use `'snd '` format 1 or format 2 for creating sound resources?

A: The format 1 `'snd '` is much more versatile. It can be used in the `_SndPlay` routine for any synthesizer and requires minimal programming effort. There is no recommendation for using either format. A format 1 has more advantages, and may contain everything a format 2 does. A format 2 is for a sampled sound only.

Q: I've opened a channel for the sampled sound synthesizer and I'm using `_SndPlay`. After awhile the system either hangs or crashes. What's wrong?

A: This is the most common abuse of the Sound Manager. The `'snd '` being used has specified a `'snth'` resource (a format 2 `'snd '` is assumed for the sampled sound synthesizer). The Sound Manager will attempt to link this `'snth'` to the channel with every call to `_SndPlay`. What's wrong is that the synthesizer has already been installed and the Sound Manager is attempting to install it again, only this time as a modifier. The same `'snth'` code has been install more than once in the channel. If the `'snd '` contains `'snth'` information, then `_SndPlay` can be used once and only once on a channel. There two possible solutions: Do the pointer math to obtain the sampled sound header and use the `bufferCmd`, or dispose of the channel after each call to `_SndPlay`.

Q: How can I use a sampled sound to play a sequence of notes?

A: Begin by opening a sampled sound channel. Load and lock the `'snd '` resource containing the sample sound into memory. Then obtain a pointer to the sampled sound header. Pass this pointer to the channel using the `soundCmd`. Now the sound is installed and ready for a sequence of `noteCmd`s. This sampled sound must contain an ending loop point or the `noteCmd` may not be heard.

Q: How do I change the play back rate of a sampled sound? Do I use the `freqCmd` or the `rateCmd`?

A: It is possible to change the `sampling rate` contained in the sampled sound header and then use the `bufferCmd`. The `freqCmd` currently requires decimal note values and will not support real frequency values. The `rateCmd` will only affect a sound that is currently in progress and is used for pitch bending effects. It is possible to add a few bytes of silence to the beginning of the sample to allow the `rateCmd` enough time to adjust the play back rate without hearing the bending affect on its pitch.

Q: How can I play multiple sampled sounds to play as a single sampled sound without the glitch that is heard between each sample on the Mac Plus?

A: On the Macintosh Plus or SE, the Sound Manager uses a 370 byte buffer internally to play sampled sounds. If the array of sampled sound data is in multiples of 370 bytes, the Sound Manager will not have to pad its internal buffer with silence. Using double buffering techniques, an application can send multiple sampled sounds using the `bufferCmd` from a `CallBack` procedure to create a continuous sound. Use this technique until the `continueCmd` is supported.

Q: How can I use the MIDI synthesizers with my own keyboards?

A: They have too many limitations at this time. Don't bother trying.

NOTE VALUES AND DURATIONS

Tempo in beats/min	30	60	90	120	150	180
𝅝 whole note	16000	8000	5333	4000	3200	2667
𝅗𝅥 half note	8000	4000	2667	2000	1600	1333
♩ dotted quarter note	6000	3000	2000	1500	1200	1000
♩ quarter note	4000	2000	1333	1000	800	667
♪. dotted eighth note	3000	1500	1000	750	600	500
♪ eighth note	2000	1000	667	500	400	333
𝅘𝅥𝅯. dotted sixteenth note	1500	750	500	375	300	250
𝅘𝅥𝅯 sixteenth note	1000	500	333	250	200	167

Table 3 duration values

Table 3 shows the duration values that are used in a `waitCmd`, `howOftenCmd`, `wakeUpCmd`, `noteCmd`, and `restCmd`. Their `duration` is in half-millisecond values. This chart will help in determining the actual `duration` used in certain tempos. To calculate the `duration` use the following formula.

```
duration = (2000/(beats per minute/60)) * beats per note
```

To calculate the `duration` for a note at a given tempo, divide the beats per minute by 60 to get the number of beats per second. Then divide the beats per second into 2000, which is the number of half-milliseconds in a second. Multiply this ratio with the number of beats the note should receive. For example, in a 4/4 time signature each sixteenth note receives 1/4th of a beat. If an application is playing a song in 120 beats per minute and wanted four sixteenth notes, each `noteCmd` would have a `duration` of 250.

	A	A#	B	C	C#	D	D#	E	F	F#	G	G#
Octave 1					1	2	3	4	5	6	7	8
Octave 2	9	10	11	12	13	14	15	16	17	18	19	20
Octave 3	21	22	23	24	25	26	27	28	29	30	31	32
Octave 4	33	34	35	36	37	38	39	40	41	42	43	44
Octave 5	45	46	47	48	49	50	51	52	53	54	55	56
Octave 6	57	58	59	**60**	61	62	63	64	65	66	67	68
Octave 7	69	70	71	72	73	74	75	76	77	78	79	80
Octave 8	81	82	83	84	85	86	87	88	89	90	91	92
Octave 9	93	94	95	96	97	98	99	100	101	102	103	104
Octave 10	105	106	107	108	109	110	111	112	113	114	115	116
Octave 11	117	118	119	120	121	122	123	124	125	126	127	

Table 4 noteCmd values

Table 4 shows the values that can be sent with a `noteCmd`. Middle C is represented by a value of 60. These values correspond to MIDI note values.

SUMMARY OF THE SOUND MANAGER

Sound Manager Constants

```
{ sound command numbers }
nullCmd            = 0;  {utility generally sent by Sound Manager}
initCmd            = 1;  {utility generally sent by Sound Manager}
freeCmd            = 2;  {utility generally sent by Sound Manager}
quietCmd           = 3;  {utility generally sent by Sound Manager}
flushCmd           = 4;  {utility generally sent by Sound Manager}
waitCmd            = 10; {sync control sent by application or modifier}
pauseCmd           = 11; {sync control sent by application or modifier}
resumeCmd          = 12; {sync control sent by application or modifier}
callBackCmd        = 13; {sync control sent by application or modifier}
syncCmd            = 14; {sync control sent by application or modifier}
emptyCmd           = 15; {sync control sent by application or modifier}
tickleCmd          = 20; {utility sent by Sound Manager or modifier}
requestNextCmd     = 21; {utility sent by Sound Manager or modifier}
howOftenCmd        = 22; {utility sent by Sound Manager or modifier}
wakeUpCmd          = 23; {utility sent by Sound Manager or modifier}
availableCmd       = 24; {utility sent by application}
versionCmd         = 25; {utility sent by application}
noteCmd            = 40; {basic command supported by all synthesizers}
restCmd            = 41; {basic command supported by all synthesizers}
freqCmd            = 42; {basic command supported by all synthesizers}
ampCmd             = 43; {basic command supported by all synthesizers}
timbreCmd          = 44; {noteSynth only}
waveTableCmd       = 60; {waveTableSynth only}
phaseCmd           = 61; {waveTableSynth only}
soundCmd           = 80; {sampledSynth only}
bufferCmd          = 81; {sampledSynth only}
rateCmd            = 82; {sampledSynth only}
continueCmd        = 83; {sampledSynth only}

{ synthesizer resource IDs used with _SndNewChannel }
noteSynth          = 1;        {note synthesizer}
waveTableSynth     = 3;        {wave table synthesizer}
sampledSynth       = 5;        {sampled sound synthesizer}
midiSynthIn        = 7;        {MIDI synthesizer in}
midiSynthOut       = 9;        {MIDI synthesizer out}

{ init options used with _SndNewChannel }
initChanLeft       = $02;      {left channel - sampleSynth only}
initChanRight      = $03;      {right channel- sampleSynth only}
initChan0          = $04;      {channel 0 - wave table only}
initChan1          = $05;      {channel 1 - wave table only}
initChan2          = $06;      {channel 2 - wave table only}
initChan3          = $07;      {channel 3 - wave table only}
initSRate22k       = $20;      {22k sampling rate - sampleSynth only}
initSRate44k       = $30;      {44k sampling rate - sampleSynth only}
initMono           = $80;      {monophonic channel - sampleSynth only}
initStereo         = $C0;      {stereo channel - sampleSynth only}
```

```
stdQLength         = 128;       {channel length for holding 128 commands}

{ sample encoding options }
stdSH              = $00        {standard sound header}
extSH              = $01        {extended sound header}
cmpSH              = $02        {compressed sound header}

{ Sound Manager error codes }
noErr              =    0       {no error}
noHardware         = -200       {no hardware to support synthesizer}
notEnoughHardware  = -201       {no more channels to support synthesizer}
queueFull          = -203       {no room left in the channel}
resProblem         = -204       {problem loading the resource}
badChannel         = -205       {invalid channel}
badFormat          = -206       {handle to snd resource was invalide}
```

Sound Manager Data Types

```
Time          = LONGINT;

SndCommand    = PACKED RECORD
                  cmd:            INTEGER; {command number}
                  param1:         INTEGER; {first parameter}
                  param2:         LONGINT; {second parameter}
               END;

ModifierStubPtr = ^ModifierStub;
ModifierStub = PACKED RECORD
                  nextStub:       ModifierStubPtr; {pointer to next stub}
                  code:           ProcPtr;     {pointer to modifier}
                  userInfo:       LONGINT;     {free for modifier's use}
                  count:          Time;        {used internally}
                  every:          Time;        {used internally}
                  flags:          SignedByte;  {used internally}
                  hState:         SignedByte;  {used internally}
               END;


SndChannelPtr = ^SndChannel;
SndChannel = PACKED RECORD
                  nextChan:       SndChannelPtr; {pointer to next channel}
                  firstMod:       ModifierStubPtr; {ptr to first modifier}
                  callBack:       ProcPtr;     {ptr to call back procedure}
                  userInfo:       LONGINT;     {free for application's use}
                  wait:           Time;        {used internally}
                  cmdInProgress:  SndCommand;  {used internally}
                  flags:          INTEGER;     {used internally}
                  qLength:        INTEGER;     {used internally}
                  qHead:          INTEGER;     {used internally}
                  qTail:          INTEGER;     {used internally}
                  queue:          ARRAY [0..stdQLength-1] OF SndCommand;
               END;
```

```
SoundHeaderPtr = ^SoundHeader;
SoundHeader = PACKED RECORD            {sampled sound header}
               samplePtr:     Ptr;     {NIL if samples in sampleArea}
               length:        LONGINT; {number of samples in array}
               sampleRate:    Fixed;   {sampling rate}
               loopStart:     LONGINT; {loop point beginning}
               loopEnd:       LONGINT; {loop point ending}
               encode:        BYTE;    {sample's encoding option}
               baseNote:      BYTE;    {base note of sample}
               sampleArea:    PACKED ARRAY [0..0] OF Byte;
            END;


{refer to the Audio Interchange File Format “AIFF” specification}
ExtSoundHeaderPtr = ^ExtSoundHeader;
ExtSoundHeader = PACKED RECORD         {extended sample header}
               samplePtr:     Ptr;     {NIL if samples in sampleArea}
               length:        LONGINT; {number of sample frames}
               sampleRate:    Fixed;   {rate of original sample}
               loopStart:     LONGINT; {loop point beginning}
               loopEnd:       LONGINT; {loop point ending}
               encode:        BYTE;    {sample's encoding option}
               baseNote:      BYTE;    {base note of original sample}
               numChannels:   INTEGER; {number of chans used in sample}
               sampleSize:    INTEGER; {bits in each sample point}
               AIFFSampleRate:EXTENDED;{rate of original sample}
               MarkerChunk:   Ptr;     {pointer to a marker info}
               InstrumentChunks:Ptr;   {pointer to instrument info}
               AESRecording:  Ptr;     {pointer to audio info}
               FutureUse1:    LONGINT;
               FutureUse2:    LONGINT;
               FutureUse3:    LONGINT;
               FutureUse4:    LONGINT;
               sampleArea:    PACKED ARRAY [0..0] OF Byte;
            END;
```

Sound Manager Routines

```
FUNCTION SndDoCommand    (chan: SndChannelPtr; cmd: SndCommand;
                          noWait: BOOLEAN): OSErr;
      INLINE $A803;

FUNCTION SndDoImmediate (chan: SndChannelPtr; cmd: SndCommand): OSErr;
      INLINE $A804;

FUNCTION SndNewChannel   (VAR chan: SndChannelPtr; synth: INTEGER;
                          init: LONGINT; userRoutine: ProcPtr): OSErr;
      INLINE $A807;

FUNCTION SndDisposeChannel (chan: SndChannelPtr;
                            quietNow: BOOLEAN): OSErr;
      INLINE $A801;

FUNCTION SndPlay         (chan: SndChannelPtr; sndHdl: Handle;
                          async: BOOLEAN): OSErr;
      INLINE $A805;

FUNCTION SndControl      (id: INTEGER; VAR cmd: SndCommand): OSErr;
      INLINE $A806;

FUNCTION SndAddModifier (chan: SndChannelPtr; modifier: ProcPtr;
                          id: INTEGER; init: LONGINT): OSErr;
      INLINE $A802;

PROCEDURE MyCallBack     (chan: SndChannelPtr; cmd: SndCommand);

FUNCTION MyModifier      (chan: SndChannelPtr; VAR cmd: SndCommand;
                          mod: ModifierStub): BOOLEAN;
```

INDEX

THE SCRIPT MANAGER 2.0

Revised by: John Harvey February 1989
Written by: Mark Davis & Sue Bartalo March 1988

This is an additional chapter for *Inside Macintosh* which documents version 2.0 of the Script Manager. This chapter includes extended date and time utility routines, general-purpose number formatting routines, and additional text manipulation routines.
Changes since October 1988: Fixed minor inaccuracies and added C examples.

Overview

The Script Manager 2.0 release extends the tools and capabilities of developers on the Macintosh for three areas: text, dates and numbers. In addition, some minor bugs were fixed and performance enhancements incorporated.

The new text routines include: lexically interpreting different scripts (e.g., in macro languages); allotting justification to different format runs within a line; ordering format runs properly with bidirectional text (Hebrew & Arabic); quickly separating Roman from non-Roman text, and determining word-wrap in text processing. The international utilities text comparison routines were significantly improved in performance, in amounts ranging from 25% to 94%.

The Macintosh date routines are extended to provide a larger range (roughly 35 thousand years), and more information. This extension allows programs that need a larger range of dates to use system routines rather than produce their own, which may not be internationally compatible. The programmer can also access the stored location (latitude and longitude) and time zone of the Macintosh from parameter RAM. The Map cdev gives users the ability to change and reference these values.

The new number routines supplement SANE, allowing applications to display formatted numbers in the manner of Microsoft Excel or Fourth Dimension, and to **read** both formatted and simple numbers. The formatting strings allow natural display and entry of numbers and editing of format strings even though the original numbers **and** the format strings were entered in a language other than that of the final user.

Implementation Notes

Some of the following routines have parameter blocks with reserved fields. **These fields must be zeroed.**

In general, the additional routines are handled by the Script Manager rather than script interface systems. The three exceptions are `FindScriptRun`, `PortionText`, and `VisibleLength` which are handled by the individual script systems (such as Roman). The version of the Script Manager can be checked before using any of these routines, to make sure that it is Script Manager 2.0 (version is $0200 or greater). For compatibility, all Script Systems test the version of the Script Manager and do not initialize if the major version number (first byte) is greater than they expect.

For testing only, the version number in INIT 2 can be changed in ResEdit in the resource header to enable those systems to run; the header has the following format:

60xx	Branch
xxxx	Flags word
4943	Resource type (INIT)
4954	
0002	Resource number (2)
02xx	Script Manager version: change to 01FF for testing

For an old script, the three routines `FindScriptRun`, `PortionText`, and `VisibleLength` will not work at all. In addition, the `'itl4'` resource (see below) for the script will not be present, so the `IntlTokenize` and number formatting routines will not work properly for the particular script's features.

The results returned from the new function calls are error and status codes which are found in the MPW 3.0 header and interface files.

Note that in the following text, the term "language" generally refers to a natural language rather than a programming language.

'Itl4' Resource

There is a new international resource, `'itl4'`, which contains information used by several of these routines and must be localized for each script (including Roman).

In Pascal:

```
Itl4Rec        = RECORD
               flags:              integer;
               resourceType:       longInt;
               resourceNum:        integer;
               version:            integer;
               resHeader1:         longInt;
               resHeader2:         longInt;
               numTables:          integer;       { one-based }
               mapOffset:          longInt;       { offsets are from record start }
               strOffset:          longInt;
               fetchOffset:        longInt;
               unTokenOffset:      longInt;
               defPartsOffset:     longInt;
               resOffset6:         longInt;
               resOffset7:         longInt;
               resOffset8:         longInt;
                                                  { the rest is data pointed to by offsets}
END;

Itl4Ptr =      ^Itl4Rec;
Itl4Handle =   ^Itl4Ptr;
```

In C:

```
struct Itl4Rec {
    short flags;
    long resourceType;
    short resourceNum;
    short version;
    long resHeader1;
    long resHeader2;
    short numTables;                    /*one-based*/
    long mapOffset;                     /*offsets are from record start*/
    long strOffset;
    long fetchOffset;
    long unTokenOffset;
    long defPartsOffset;
    long resOffset6;
    long resOffset7;
    long resOffset8;
};

#ifndef __cplusplus
typedef struct Itl4Rec Itl4Rec;
#endif

typedef Itl4Rec *Itl4Ptr, **Itl4Handle;
```

Text

The new text routines include: lexically interpreting different scripts (e.g., in macro languages); allotting justification to different format runs within a line; ordering format runs properly with bidirectional text (Hebrew & Arabic); quickly separating Roman from non-Roman text, and determining word-wrap in text processing. The international utilities text comparison routines were significantly improved in performance, in amounts ranging from 25% to 94%.

Parse Table

In Pascal:

```
Type
       CharByteTable = Packed Array [0..255] of SignedByte;

Function ParseTable(table: CharByteTable): Boolean;

typedef char CharByteTable[256];
```

In C:

```
pascal Boolean ParseTable(CharByteTable table);
```

Double-byte characters have distinctive high (first) bytes, which allows them to be distinguished from single-byte characters. The `ParseTable` routine can be used to traverse double-byte text quickly. It does this by filling a table of bytes with values which indicate the extra number of bytes taken by a given character. This array can then be used instead of making function calls on each byte. As with the other script-specific routine calls, the values in the table will vary with the script of the current font in `thePort`, so you must make sure to set the font correctly.

An entry in the table is set to 0 for a single-byte character and 1 for the first byte of a double-byte character. (With a single-byte script, the entries are all zero.) The return value from the routine will always be true. This routine has always been present in the Script Manager, but was not documented until now. Also note that script systems will never require more than two bytes per character, so you can safely assume that there are only single-byte and double-byte characters.

For example, in the following code the reference to `tablePtr[myChar]` is functionally equivalent to a use of `_CharByte`, but does not involve a trap call.

In Pascal:

```
Var
        myChar:         Integer;
        i, max:         Integer;
        tablePtr:       CharByteTable;
        s:              String [255];
        parseResult:    Boolean;

Begin
        parseResult := ParseTable(tablePtr);
        i := 1;
        max := length (s);
        While i <= max do Begin
                myChar := ord(s[i]);                            {get byte}
                i := i + 1;                                     {skip to start of next}
                if (tablePtr[myChar] <> 0) then Begin           {if double-byte}
                        myChar := myChar * $100 + ord(s[i]);{include next byte}
                        i := i + 1;                             {skip to start of next}
                End;
                {do something with myChar}
        End;
End;
```

In C:

```
short           mychar;
CharByteTable table;
char            *s = "Test String";
Boolean         parseResult;

{
        parseResult = ParseTable(table);

        while ( *s ) {
                mychar = *s;                            /*get the first byte*/
                s++;
                if ( table[*s] <> 0 )
                        mychar = (mychar * 0x100) + *s;
                /* Do something with mychar */
        }
}
```

Remember that the `CharByteTable` is specific to the script. There could be two or three scripts installed that are double-byte and have different `CharByteTable` arrays.

IntlTokenize

In Pascal:

```
Function IntlTokenize ( tokenParam : TokenBlockPtr  ): TokenResults;
```

In C:

```
pascal TokenResults IntlTokenize(TokenBlockPtr tokenParam);
```

The `IntlTokenize` routine is intended for use in macro expressions and similar programming constructs intended for general users. It allows the program to recognize variables, symbols and quoted literals without depending on the particular natural language (e.g., English vs. Japanese).

The routine is a mildly programmable regular expression recognizer for parsing text into tokens. The single parameter is a parameter block describing the text to be tokenized, the destination of the token stream, the `'itl4'` resource handle, and the various programmable options. `IntlTokenize` will return a list of tokens found in the text.

In Pascal:

```
       TokenBlock = RECORD
              source: Ptr;                 {pointer to stream of characters}
              sourceLength: LongInt;       {length of source stream}
              tokenList: Ptr;              {pointer to array of tokens}
              tokenLength: LongInt;        {maximum length of TokenList}
              tokenCount: LongInt;         {number of tokens generated by tokenizer}
              stringList: Ptr;             {pointer to stream of identifiers}
              stringLength: LongInt;       {length of string list}
              stringCount: LongInt;        {number of bytes currently used}
              doString: Boolean;           {make strings & put into StringLIst}
              doAppend: Boolean;           {append to TokenList rather than replace}
              doAlphanumeric: Boolean;     {identifiers may include numeric}
              doNest: Boolean;             {do comments nest?}
              leftDelims, rightDelims: ARRAY[0..1] OF TokenType;
              leftComment, rightComment: ARRAY[0..3] OF TokenType;
              escapeCode: TokenType;       {escape symbol code}
              decimalCode: TokenType;      {decimal symbol code}
              itlResource: Handle;         {itl4 resource handle of current script}
              reserved: array [0..7] of Longint;  { must be zeroed! }
       END;

TokenType = Integer;          {see list of TokenType values at end of document}
TokenRec = RECORD
       theToken:              TokenType;
       position:              Ptr;                  {ptr into original source}
       length:                LongInt;              {length of text in original source}
       stringPosition:        StringPtr;            {Pascal/C string copy of identifier}
END;
```

In C:

```
struct TokenBlock {
    Ptr source;                    /*pointer to stream of characters*/
    long sourceLength;             /*length of source stream*/
    Ptr tokenList;                 /*pointer to array of tokens*/
    long tokenLength;              /*maximum length of TokenList*/
    long tokenCount;               /*number tokens generated by tokenizer*/
    Ptr stringList;                /*pointer to stream of identifiers*/
    long stringLength;             /*length of string list*/
    long stringCount;              /*number of bytes currently used*/
    Boolean doString;              /*make strings & put into StringLIst*/
    Boolean doAppend;              /*append to TokenList rather than replace*/
    Boolean doAlphanumeric;        /*identifiers may include numeric*/
    Boolean doNest;                /*do comments nest?*/
    TokenType leftDelims[2];
    TokenType rightDelims[2];
    TokenType leftComment[4];
    TokenType rightComment[4];
    TokenType escapeCode;          /*escape symbol code*/
    TokenType decimalCode;
    Handle itlResource;            /*ptr to itl4 resource of current script*/
    long reserved[8];              /*must be zero!*/
};

#ifndef __cplusplus
typedef struct TokenBlock TokenBlock;
#endif

typedef TokenBlock *TokenBlockPtr;

typedef short TokenType;

struct TokenRec {
    TokenType theToken;
    Ptr position;                  /*pointer into original Source*/
    long length;                   /*length of text in original source*/
    StringPtr stringPosition;      /*Pascal/C string copy of identifier*/
};
```

For the `TokenBlock` record:

`source` is a pointer to the beginning of a stream of characters (not a Pascal string).

`sourceLength` is the number of characters in the source stream.

`tokenList` is a pointer to memory allocated by the application for the token stream. The tokenizer places the tokens it generates at and after the address in `tokenList`.

`tokenLength` is the number of tokens that will fit in the memory pointed to by `tokenList` (not the number of bytes).

`tokenCount` is the number of tokens that are currently occupying the space pointed to by `tokenList`. If the `doAppend` flag is true, then `tokenCount` must be a correct number before calling the tokenizer. The tokenizer modifies this value to show how many tokens are in the token stream after tokenizing.

`stringList` is a pointer to memory allocated by the application for strings that the tokenizer generates if the `doString` flag is true. If the flag is false, then `stringList` is ignored.

stringLength is the number of bytes of memory allocated for stringList.

stringCount is the number of bytes that are currently occupying the space pointed to by stringList. If the doAppend flag is true, then stringCount must be a correct number before calling the tokenizer. The tokenizer modifies this value to show how many bytes are in the string stream after tokenizing.

doString is a boolean flag that instructs the tokenizer to create a sequence of even-boundaried, null-terminated Pascal strings. Each token generated by the tokenizer will have a string created to represent it if the flag is true. Each token record contains the address of the string that represents it.

doAppend is a boolean flag that instructs the tokenizer to append tokens to the space pointed to by tokenList rather than replace whatever is there. tokenCount must correctly reflect the number of tokens in the space pointed to by tokenList.

doAlphanumeric is a boolean flag that, when true, states that numerics may be mixed with alphabetics to create alphabetic tokens.

doNest is a boolean flag that instructs the tokenizer to allow nested comments of any depth.

leftDelims is an array of two integers, each of which corresponds to the class of the symbol that may be used as a left delimiter for a quoted literal. Double quotes, for instance, is class token2Quote. If only one left delimiter is needed, the other must be specified to be delimPad.

rightDelims is an array of two integers, each of which corresponds to the class of the symbol that may be used as the matching right delimiter for the corresponding left delimiter in leftDelims.

leftComment is an array of four integers. Each successive pair of two describes a pair of tokens that may be used as left delimiters for comments. These tokens are stored in reverse order. The tokens numbered zero and two are the second tokens of the two-token sequences; the tokens numbered one and three are the first tokens of the two-token sequences.

If only one token is needed for a delimiter, the second token must be specified to be delimPad. If only one delimiter is needed, then both of the tokens allocated for the other symbol must be delimPad. The first token of a two-token sequence is the higher position in the array. For example, the two left delimiters (* and { would be specified as

```
leftComment[0]:= tokenAsterisk      (*asterisk*)
leftComment[1]:= tokenLeftParen;    (*left parenthesis*)
leftComment[2]:= delimPad ;         (*nothing*)
leftComment[3]:= tokenLeftCurly;    (*curly brace*)
```

rightComment is an array of four integers with similar characteristics as leftComment. The positions in the array of the right delimiters must be the same as their matching left delimiters.

escapeChar is a single integer that is the class of the symbol that may be used for an escape character. The tokenizer considers the escape character to be an escape character (as opposed to being itself) only within quoted literals.

If backslash (\) is given as the escapeChar, then the tokenizer would consider it an escape character in the following string:

```
"This is an escape\n"
```

It would not be considered an escape character in a non-quoted string like the following:

```
This isn't an escape\n
```

decimalCode is a single integer that is the tokenType that may be used for a decimal point. The tokenizer considers the decimal character to be a decimal character (as opposed to being itself) only when flanked by numeric or alternate numeric characters, or when following them. When the strings option is selected, the decimal character will always be transliterated to an ASCII period (and alternate numbers will be transliterated to ASCII digits).

itlResource is a handle to the 'itl4' resource of the script in current use. The application must load the 'itl4' resource and place its handle here before calling the tokenizer. Every time the script of the text to be tokenized changes, the pointer to the respective 'itl4' resource must be placed here.

reserved locations must all be zeroed.

For the token record:

theToken is the ordinal value of the token represented by the token record.

position points to the first character in the original text that caused this particular token to be generated.

length is the length in bytes of the original text corresponding to this token.

stringPosition points to a null-terminated, even-boundaried Pascal string that is the result of using the doString option. If doString is false then stringPosition is always set to NIL.

The available token types are: whitespace, newline, alphabetic, numeric, decimal, endOfStream, unknown, alternate numeric, alternate decimal, and a host of fixed token symbols, such as (# @ : := .

The tokenizer does not attempt to provide complete lexical analysis, but rather offers a programmable "pre-lex" function whose output should then be processed by the application at a lexical or syntactic level.

The programmable options include: whether to generate strings which correspond to the text of each token; whether the current tokenize call is to append to, rather than replace, the current token list; whether alphabetic tokens may have numerics within them; whether comments may be nested; what the left and right delimiters for comments are (up to two sets may be specified); what the left and right delimiters for quoted literals are (up to two sets may be specified); what the escape character is; and what the decimal point symbol is.

Some users may use two or more different scripts within a program. However, each script's character stream must be passed separately to the tokenizer because different resources must be passed to the tokenizer depending on the script of the text stream. Appending tokens to the token stream lets the application see the tokens generated by the different scripts' characters as a single token stream. Restriction: users may not change scripts within a comment or quoted literal because these syntactic units must be complete within a single call to the tokenizer in order to avoid tokenizer syntax errors.

The application may specify up to two pairs of delimiters each for both quoted literals and comments. Quoted literal delimiters consist of a single symbol, and comment delimiters may be either one or two symbols (including newline for notations whose comments automatically terminate at the end of lines). The characters that compose literals within quoted literals and comments are normally defined to have no syntactic significance; however, the escape character within a quoted literal does signal that the following character should not be treated as the right delimiter. Each delimiter is represented by a token, as is the literal between left and right delimiters.

If two different comment delimiters are specified by the application, then the `doNest` flag always applies to both. Comments may be nested if so specified by the `doNest` flag with one restriction that must be strictly observed in order to prevent the tokenizer from malfunctioning: nesting is legal only if **both** the left and right delimiters for the comment token are composed of two symbols each. In this version, there is limited support for nested comments. When using this feature, test to insure that it meets your requirements.

An escape character between left and right delimiters of a quoted literal signals that the following character is not the right delimiter. An escape character is not specially recognized and has no significance outside of quoted literals. When an escape character is encountered, the portion of the literal before the escape is placed into a single token, the escape character itself becomes a token, the character following the escape becomes a token, and the portion of the literal following the escape sequence becomes a token.

A sequence of whitespace characters becomes a single token.

Newline, or carriage return, becomes a single token.

A sequence of alphabetic characters becomes an alphabetic token. If the `doAlphanumeric` flag is set, then alphabetic characters include digits, but the first character must be alphabetic.

A sequence of numeric characters becomes a numeric token.

A sequence of numeric characters followed by a decimal mark, and optionally followed by more numeric characters, becomes a realNumber token.

Some scripts have not only “English” digits, but also their own numeral codes, which of course will be unrecognizable to the typical application. A sequence of alternate digits becomes an alternate numeric token. If the strings option is selected then the digits will be transliterated to “English” digits. This includes the realNumber tokens, whose results become alternate real tokens.

The end of the character stream becomes a token.

A token record consists of a token code, a pointer into the source stream (signifying the first character of the sequence that generated the token), the byte length of the sequence of characters that generated the token, and space for a pointer to a Pascal string, explained next.

The application may instruct the tokenizer to generate null-terminated, even-boundaried Pascal strings corresponding to each token. In this case, if the token is anything but alphabetic or numeric then the text of the source stream is copied verbatim into the Pascal string. Otherwise, if the text in the source stream is Roman letters or numbers then those characters are transliterated into Macintosh eight-bit ASCII and a string is created from the result, allowing users of other languages to transparently use their own script's numerals or Roman characters for numbers or keywords. Non-Roman alphabetics are copied verbatim.

Semantic attributes of byte codes vary from natural language to natural language. As an example, in the Macintosh character set code $81 is an Å, but in Kanji this code is the first byte of many double-byte characters, some of which are alphabetic, some numeric, and some symbols. This information is retrieved from the 'itl4' resource, which also contains a canonical string format for the fixed tokens, so that the internal format of formulæ can be redisplayed in the original language.

'itl4' also holds a string copy routine which converts the native text to the corresponding English (except for alphanumerics). As with the other international resources, the choice of 'itl4' depends on the script interface system in use.

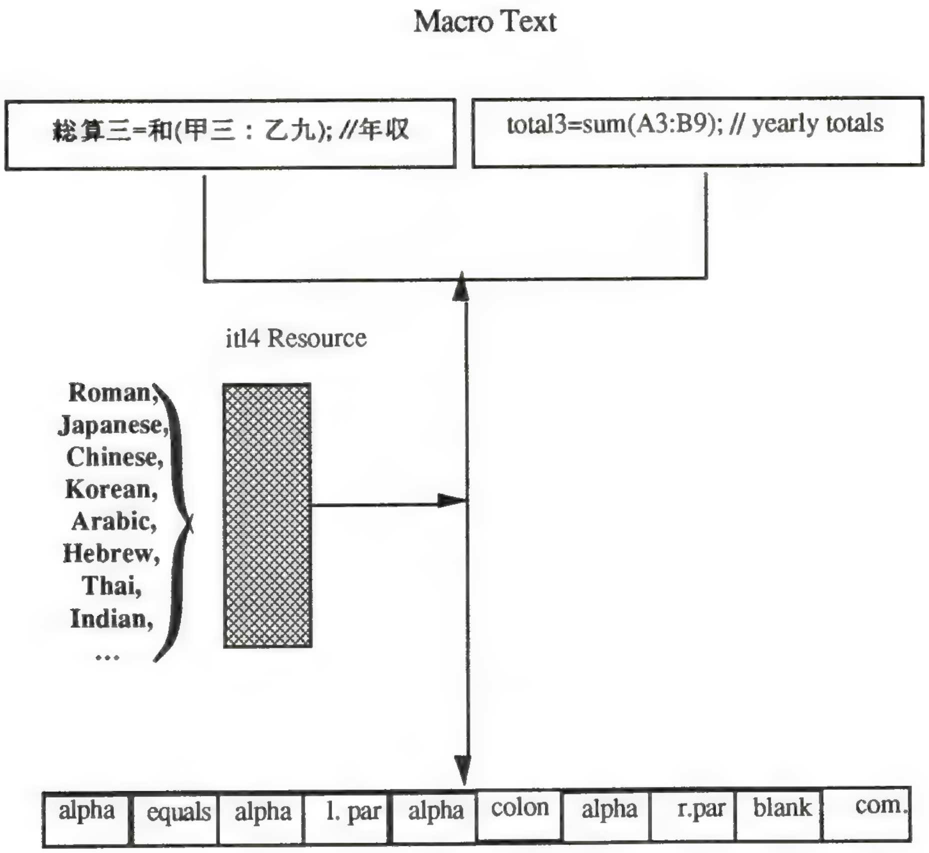


Figure 1–IntlTokenize

The untokenTable in the 'itl4' resource contains standard representations for the fixed tokens, and can be used to display the internal format. An example of how a user might access this table and use the token information follows:

In Pascal:

```
Type
        UntokenTable = Record
                len: Integer;
                lastToken: Integer;
                index: array [0..255] of Integer; {index table; last = lastToken}
                {list of pascal strings here. index pointers are from front of table}
        End;
        UntokenTablePtr = ^UntokenTable;
        UntokenTableHandle = ^UntokenTablePtr;

Function GetUntokenTable( Var x: UntokenTableHandle ): Boolean;
Var
        itl4: itl4Handle;
        p: UntokenTablePtr;
Begin
        GetUntokenTable := false;                       {assume error}
        itl4 := itl4Handle(IUGetIntl(4));               {get itl4 record}
        if itl4 <> nil then begin                       {if ok}
                HLock(Handle(itl4));                    {lock for safety}
                p := UntokenTablePtr(ord(itl4^)+itl4^^.untokenOffset);
                                                        {untokenize parts subtable}
                With p^ Do Begin                        {using resource table}
                        x := UntokenTableHandle(NewHandle(len));
                                                        {make handle of proper size}
                        BlockMove(Ptr(p),Ptr(x^),len);  {copy contents}
                End;
                HUnlock(Handle(itl4));                  {free back up}
                GetUntokenTable := true;                {no error}
        End;
End;

If (GetUntokenTable(myUntokenTable)) then
        With curToken^ Do Case theToken OF
                {. . .}
                tokenAlpha:
                        AppendString( myVariable[i] );
                Otherwise With myUntokenTable^^, curToken^ Do Begin
                        If theToken > lastToken Then Begin
                                AppendString( '?' );
                        End Else Begin
                                sPtr := pointer(ord(@len) + index[theToken]);
                                AppendString(sPtr^);
                        End; {if}
                End; {item}
        End; {case}
```

In C:

```
struct UntokenTable {
    short len;
    short lastToken;
    short index[256];               /*index table; last = lastToken*/
};

#ifndef __cplusplus
typedef struct UntokenTable UntokenTable;
#endif

typedef UntokenTable *UntokenTablePtr, **UntokenTableHandle;

GetUntokenTable(UntokenTableHandle *x)
{
        Itl4Handle    itl4;
        UntokenTablePtr       p;

        itl4 = (Itl4Handle)IUGetIntl(4);

        if (itl4) {
                HLock((Handle)itl4);

                p = (UntokenTablePtr)( (char *)(*itl4) + ( (*itl4)->unTokenOffset ) );

                *x = (UntokenTableHandle)NewHandle(p->len);

                if (x)
                        BlockMove((Ptr)p,(Ptr)**x,p->len);

                HUnlock((Handle)itl4);

                return((short)*x);
        }
        else
                return(0);
}

if ( GetUntokenTable(myUntokenTable) )
        switch curtoken->theToken {
        /* ... */
        case tokenAlpha:
                        AppendString(myvariable[i]);
                        break;
        default:
                        if (curtoken->theToken > lastToken)
                                AppendString("?");
                        else {
                                Hlock((Handle)myUntokenTable);
                                sptr  =  (char  *)(*myUntokenTable)  +  (*myUntokenTable)-
>index[curtoken->theToken];
                                AppendString(sptr);
                                HUnlock((Handle)myUntokenTable);
                        }
                        break;
        }
```

PortionText

In Pascal:

```
Function PortionText (textPtr : Ptr; textLen : Longint): Fixed; {proportion}
```

In C:

```
pascal Fixed PortionText(Ptr textPtr,long textLen);
```

This routine returns a result which indicates the proportion of justification that should be allocated to this text when compared to other text. It is used when justifying a sequence of format runs, so that the appropriate amount of extra width is apportioned properly among them. For example, suppose that there are three format runs on a line: A, B, and C. The line needs to be widened by 11 pixels for justification. Calling `PortionText` on these format runs yields the first row in the following table:

	A	**B**	**C**	**Total**
PortionText:	5.4	7.3	8.2	20.9
Normalized:	.258	.349	remainder	1.00
Pixels (p):	2.84	3.84	remainder	11.0
Rounded (r):	3	4	remainder	11

The proportion of the justification to be allotted to A is 25.8%, so it receives 3 pixels out of 11. In general, to prevent rounding errors, $r_n = \text{round}(\Sigma_{1..n}\ p) - \Sigma_{1..n-1}\ r$ (which can be computed iteratively); e.g., r_B is round(3.84+2.84) – 3, and r_C is round(11.0) – 7.

For normal Roman text, the result is currently a function of the number of spaces in the text, the number of other characters in the text, and the font size (the raw size, not ascent + descent + leading). This may change in the future, so values should be compared at the time of execution.

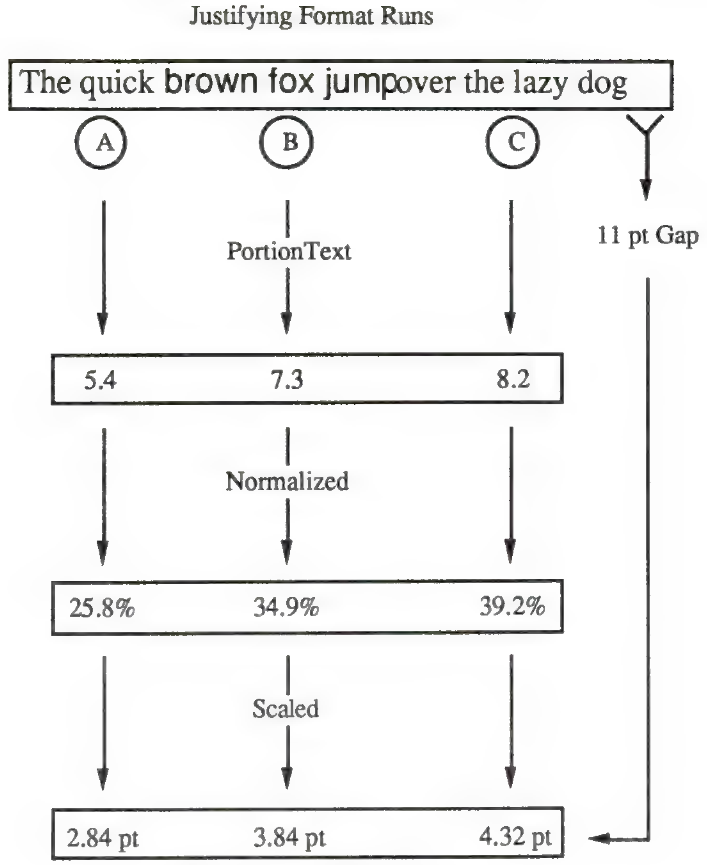


Figure 2–PortionText

Format Order

In Pascal:

```
FormatOrder = array [0..0] of Integer;
FormatOrderPtr = ^FormatOrder;

Procedure GetFormatOrder ( ordering: FormatOrderPtr;
                           firstFormat:    Integer;
                           lastFormat:     Integer;
                           lineRight:      Boolean;
                           RLDirProc:      Ptr;
                           dirParam:       Ptr);
```

In C:

```
typedef short FormatOrder[1];
typedef FormatOrder *FormatOrderPtr;

pascal void GetFormatOrder(FormatOrderPtr ordering,short firstFormat,short lastFormat,
    Boolean lineRight,Ptr rlDirProc,Ptr dirParam);
```

This routine orders the text properly for display of bidirectional format runs. Word processing programs that use this procedure for multi-font text can be independent of script text-ordering in a line (e.g., Hebrew or Arabic right-left text). The ordering points to an array of integers, with (lastFormat – firstFormat + 1) entries. The GetFormatOrder routine retrieves the direction of each format by calling the direction procedure, RLDirProc, which has the following format:

In Pascal:

```
Function MyRLDirProc ( theFormat : Integer; dirParam : Ptr) :Boolean;
```

In C:

```
pascal Boolean MyRLDirProc(short theFormat, Ptr dirParam);
```

The RLDirProc is called with the values from firstFormat to lastFormat to determine the directions of each of the format runs. It returns true for right-left text direction, otherwise false. The parameter dirParam is available to provide other necessary information for the direction procedure (i.e., style number, pointer to style array, etc).

GetFormatOrder returns a permuted list of the numbers from firstFormat to lastFormat. This permuted list can be used to draw or measure the text. (For more detail, see the Script Manager developers' packet). The lineRight parameter is true if the text is right-left orientation, otherwise false.

The array Ordering is created and filled by your application. The first element in the array should correspond to the parameter firstFormat, and the last element should correspond to lastFormat. GetFormatOrder loops through this array and passes each element in the array back to the RLDirProc function. Since you fill the ordering array and you write the RLDirProc, you should obviously store format runs in a way that makes the GetFormatOrder routine useable.

One obvious way to do this would be to declare a record type for format runs that allowed you to save things like font style, font ID, script number, and so on. You then could store these records in an array. When the time came to call GetFormatOrder, you would simply fill the Ordering array with the indexes that you used to access your array of format run records. GetFormatOrder would return an array which described the correct drawing order for your format runs.

Consider this example. Let uppercase letters stand for format runs that are left to right, and lowercase letters stand for right-left format runs. For example, there are two format runs in the following line.

```
1  2
ABCfed
```

With left-right line direction, the text should appear on the screen as:

```
1  2
ABCdef
```

With right-left line direction, the text should appear on the screen as:

```
2  1
fedABC
```

GetFormatOrder is used to tell you what order the format runs should be drawn in based on line direction for a particular line of text.

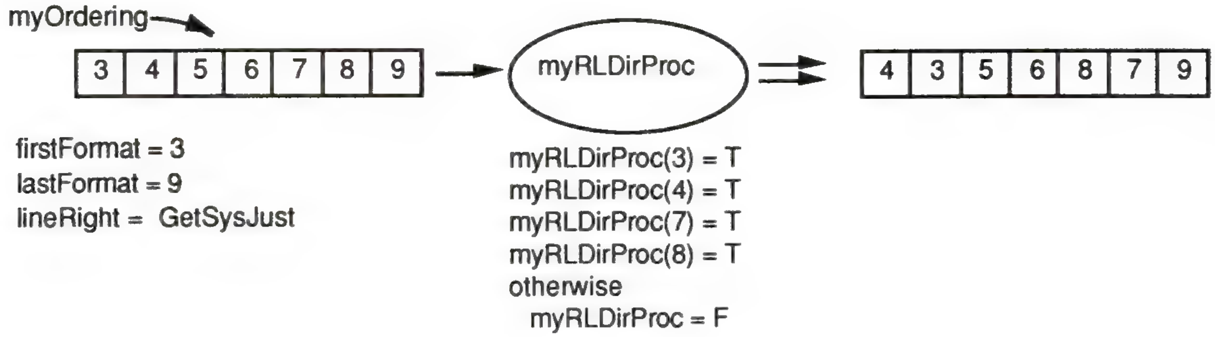


Figure 3–GetFormatOrder

For example, in Pascal:

```
GetFormatOrder(myOrdering,firstFormat,lastFormat,GetSysJust = 0,MyRLDirProc,nil);
for i := 0 to lastFormat-firstFormat do
        with MyFormat [myOrdering [i]], MyStyle [formatStyle] do begin
                TextFont(styleFont);
                {set up other text style features...}
                case what of
                        drawing: DrawText(textStartPtr, formatStart, formatLength);
                        measuring: TextWidth(textStartPtr, formatStart, formatLength);
                        {and so on}
                end; {case}
        end; {with}
end; {for}
```

In C:

```
GetFormatOrder(myOrdering,firstFormat,lastFormat,(Boolean)GetSysJust(),(Ptr)MyRLDirProc,nil);
for ( i = 0, i <= (lastFormat-firstFormat), i++)
        /* set up style stuff */
        switch what {
                case drawing:
                                DrawText(textStartPtr,formatStart,formatLength);
                                break;
                case measuring:
                                TextWidth(textStartPtr,formatStart,formatLength);
                                break;
                default:
                                break;
        }
```

FindScriptRun

In Pascal:

```
Function FindScriptRun ( textPtr: Ptr; textLen: Longint;
                         VAR lenUsed: Longint): ScriptRunStatus;

ScriptRunStatus = RECORD
       script:       SignedByte;
       variant:      SignedByte;
END;
```

In C:

```
pascal struct ScriptRunStatus FindScriptRun(Ptr textPtr,long textLen,long *lenUsed);

struct ScriptRunStatus {
    short script;
    short variant;
};

char                  *mychararray = 'abcDEFghi';
char                  *textptr;
long                  textlength;
ScriptRunStatus       srs;
long                  lenused;

srs = FindScriptRun(mychararray,(long)strlen(mychararray),&lenUsed);
/* lenUsed would now = 3, blocktype would equal 0 */
/* we can point at the remainder of the text with the following code */
textptr = mychararray + lenUsed;
textlen = strlen(mychararray) - lenUsed;
```

For compatibility, each script allows Roman text to be mixed in. This routine is used to break up mixed text (Roman & Native) into blocks. The `lenUsed` is set to reflect the length of the remaining text. The return value reflects the type of block: the upper byte is the script (0 being Roman text) and the lower byte being script-specific (script systems can return types of native sub-scripts, such as Kanji, Katakana and Hiragana for Japanese). For example, given that the capital letters represent Hebrew text:

In Pascal:

```
myCharArray = 'abcDEFghi';
myCharPtr := @myCharArray;
blockType := FindScriptRun (myCharPtr, 9, lenUsed);
{lenUsed = 3, blockType = 0: get remainder of text with: }
textPtr := ptr(ord(textPtr)+lenUsed);
textLen := textLen-lenUsed;
```

StyledLineBreak

In Pascal:

```
Function StyledLineBreak(    textPtr:       Ptr;
                             textLen:       Longint;
                             textStart:     Longint;
                             textEnd:       Longint;
                             flags: Longint;
                             Var    textWidth:     Fixed; {on exit, set if too long}
                             Var    textOffset:    Longint)
     :StyledLineBreakCode;
StyledLineBreakCode = (smBreakWord,smBreakChar,smBreakOverflow);
```

In C:

```
pascal StyledLineBreakCode StyledLineBreak(Ptr textPtr,long textLen,long textStart,
    long textEnd,long flags,Fixed *textWidth,long *textOffset);

enum {smBreakWord,smBreakChar,smBreakOverflow};
typedef unsigned char StyledLineBreakCode;
```

This routine breaks a line on a word boundary. The user will loop through a sequence of format runs, resetting the `textPtr` and `textLen` each time the script changes; and resetting the `textStart` and `textEnd` for each format run. The `textWidth` will automatically be decremented by `StyledLineBreak`.

`TextPtr` points to the start of the text, `textLen` indicates the maximum length of the text, and the `textWidth` parameter indicates the maximum pixel width of the rectangle used to display the text starting at the `textStart` and ending at the `textEnd`. The `flags` parameter is reserved for future expansion and must be zero.

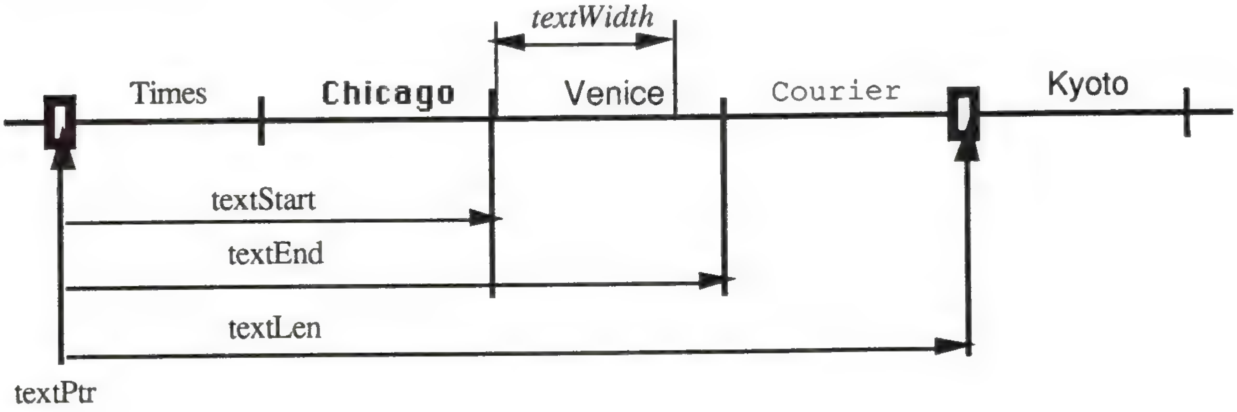


Figure 4–StyledLineBreak

On input, a non-zero `textOffset` indicates whether this is the first format run (possibly forcing a character break rather than a word break: if `textOffset` is non-zero, at least one character will be returned if the line is not empty). On output it is the number of bytes from `textPtr` up to the point where the line should be broken. If the passed `textWidth` extended beyond the end of the text (i.e., is larger than the width from `textoffset` to `textLen`), then the width of the text is subtracted from the `textWidth` and the result returned in the `textWidth` parameter. This can be used for the next format run.

The routine result indicates whether the routine broke on a word boundary, character boundary, or the width extended beyond the edge of the text.

When used with single-format text, the `textStart` can be zero, and the `textEnd` identical with the `textLen`. With multi-format text, the interval between `textStart` and `textEnd` specifies a format run. The interval between `textPtr` and `textLen` specifies a script run (a contiguous sequence of text where the script of each of the format runs is the same). Note that the format runs in `StyledLineBreak` **must** be traversed in back-end storage order, **not** display order (see `GetFormatOrder`).

In other words, if the current format run is included in a contiguous sequence of other format runs of the same script, then the `textPtr` should point to the start of the first format run of the same script, while the `textLen` should include the last format run of the same script. This is so that word boundaries can extend across format runs; they will **never** extend across script runs.

Although the offsets are in `longint` values and widths in fixed for future extensions, in the current version the `longint` values should be restricted to the integer range, and only the integer portion of the widths will be used.

VisibleLength

In Pascal:

```
FUNCTION VisibleLength ( textPtr : Ptr; textLen: Longint): Longint;
```

In C:

```
pascal long VisibleLength(Ptr textPtr,long textLen);
```

This routine returns the length of the text excluding trailing white space, taking into account the script of the text. Trailing white space is only excluded if it occurs on the visible right side, in display order.

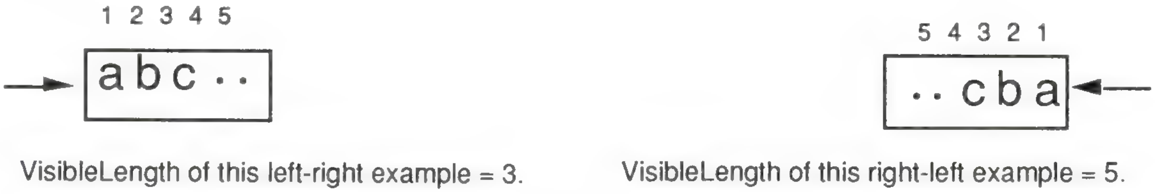

1 2 3 4 5
abc..
5 4 3 2 1
..cba
VisibleLength of this left-right example = 3.
VisibleLength of this right-left example = 5.

Figure 5–VisibleLength

For example, in Pascal:

```
    myVisibleLength := VisibleLength(myText,myOffset);
    curSlop := myPixel - TextWidth(myText,0,myVisibleLength);
    DrawJust(myText,myVisibleLength,curSlop);
```

Changing Text Case

In Pascal:

```
Procedure UprText(textPtr: Ptr; len: Integer);
Procedure LwrText(textPtr: Ptr; len: Integer);
```

In C:

```
pascal void UprText(Ptr textPtr,short len);
pascal void LwrText(Ptr textPtr,short len);
```

UprText provides a Pascal interface to the _UprString assembly routine, which will uppercase text up to 32K in length. The LwrText routine provides the corresponding lowercase routine. Both of these routines will not change the number or position of characters in a string, but are faster and simpler than the Transliterate routine.

Text Comparison

We have done some performance analyses of Pack6 comparison routines, and based upon those, were able to increase performance by about 50% on average. This increase results in a corresponding increase in 4th Dimension sorting performance, for example. Also, a long-standing bug in sorting “œ” and “æ” has been corrected. A test program on the Macintosh SE comparing “The quick brown fox jumped over the lazy dog” to variants produced the following decreases in comparison time:

Identical text:	94%
Last Character Unequal (g vs. X)	83%
Last Character Weakly Equal (g vs. G):	82%
First Character Unequal (T vs X):	59%
First Character Weakly Equal (T vs t):	29%
All Characters Weakly Equal (T vs t...g vs. G):	25%

Part of the performance increase results from internal caching of 'itl ' resources. Originally all 'itl ' resources (resulting from IUGetIntl of 0,1,2,4) were cached, but several programs do a _ReleaseResource or _DetachResource on 'itl0', rendering the cache invalid. Because of this, currently only 'itl2' and 'itl4' are cached. Developers must be sure not to release or detach these resources. Also, only the system file resources are used, so they cannot be overridden by copies in the application or document resource forks.

"The quick brown fox jumped over the lazy dog"

A. Identical "The quick brown fox jumped over the lazy dog"

Last Char

B. Unequal "The quick brown fox jumped over the lazy doX"

C. Similar "The quick brown fox jumped over the lazy doG"

First Char

D. Unequal "Xhe quick brown fox jumped over the lazy dog"

E. Similar "the quick brown fox jumped over the lazy dog"

All Chars

F. Similar "THĖ QUIÇK BRØWN FOX JUMPED ØVĖR THĖ LÅZY DOĠ"

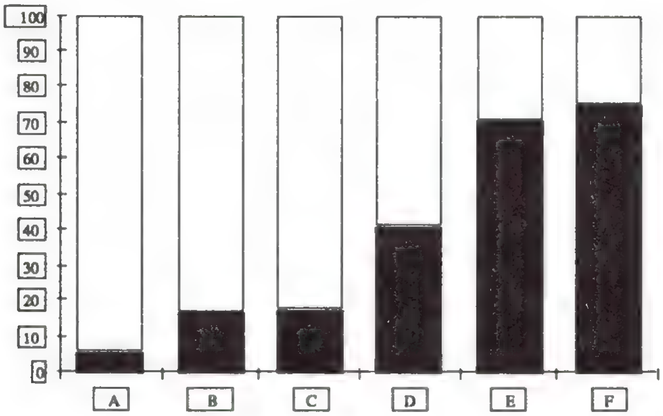


Figure 6–International Text Comparison

Dates

The Macintosh date routines are extended to provide a larger range (roughly 35 thousand years), and more information. This extension allows programs that need a larger range of dates to use system routines rather than produce their own, which may not be internationally compatible. The programmer can also access the stored location (latitude and longitude) and time zone of the Macintosh from parameter RAM. The Map cdev gives users the ability to change and reference these values.

The long internal format of a date is as before, in seconds since 12:00 midnight, January 1, 1904, but is represented as a **signed** 64-bit integer (SANE Comp format), allowing a somewhat larger range (roughly 500 billion years). Short internal format dates (since they are unsigned) can be converted to long format by filling the top 32 bits with zero; long formats can be converted to short by truncating (assuming that they are within range). When storing in files, a five (or six) byte format can be used for a range of roughly 35 thousand years. This value should be sign-extended to restore it to a Comp format.

In Pascal:

```
Type LongDateTime = Comp;
```

In C:

```
typedef comp LongDateTime;
```

The standard date conversion record is extended using a new structure:

In Pascal:

```
LongDateRec = Record
              case Integer of
              0:      (      era,year,month,day,hour,minute,second,
                             dayOfWeek,dayOfYear,weekOfYear,
                             pm,res1,res2,res3:    Integer);
              1:      (      list:  array [longDateField] of Integer);
              2:      (      eraAlt:        Integer;
                             oldDate:       DateTimeRec);
       end;
```

In C:

```
union LongDateRec {
    struct {
        short era;
        short year;
        short month;
        short day;
        short hour;
        short minute;
        short second;
        short dayOfWeek;
        short dayOfYear;
        short weekOfYear;
        short pm;
        short res1;
        short res2;
        short res3;
        } ld;
    short list[14];                    /*Index by LongDateField!*/
    struct {
        short eraAlt;
        DateTimeRec oldDate;
        } od;
};
```

The default calendar for converting to and from the long internal format is the Gregorian calendar. The `era` field for this calendar has values 0 for A.D. and -1 for B.C. (Note that the international date string conversion routines do not append strings for A.D. or B.C.) The current range allowed in conversion is roughly 30,000 BC to 30,000 AD.

(Note that in different countries the change from the Julian calendar to Gregorian calendar occurred in different years: in Catholic countries, it occurred in 1582, while in Russia it took place as late as 1917. Dates before these years in those countries should use the Julian calendar for conversion. The Julian calendar differs from the Gregorian by three days every four centuries.)

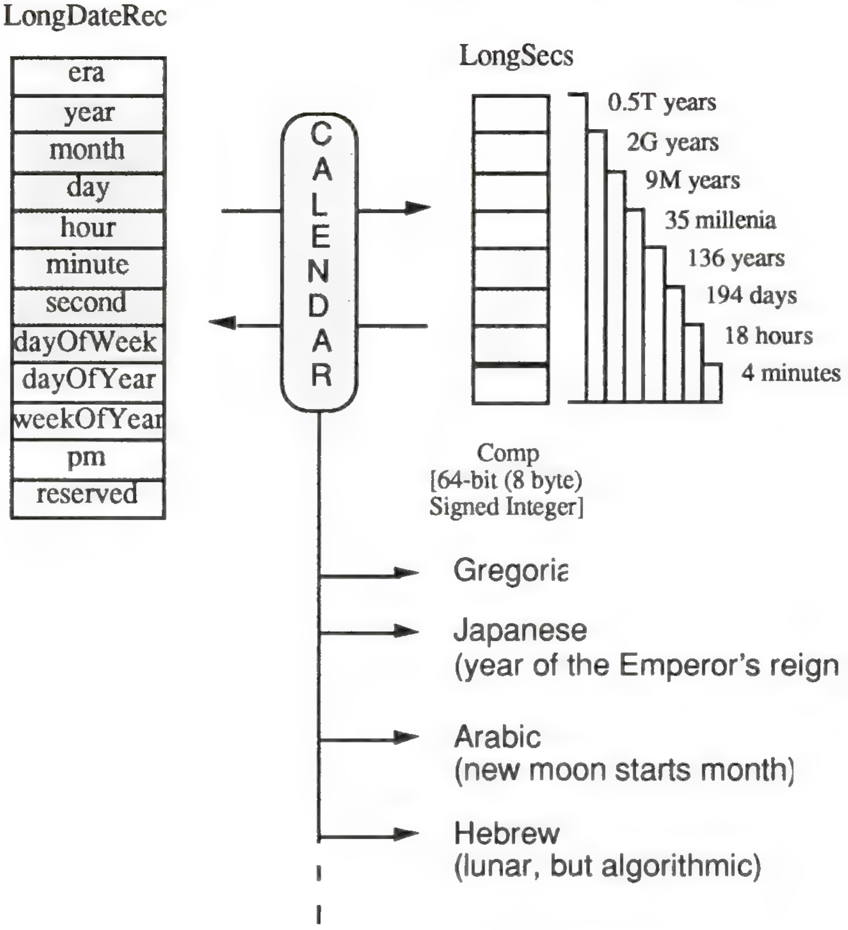


Figure 7–Long Date ´ String

InitDateCache

In Pascal:

```
Function InitDateCache (theCache: DateCachePtr): OSErr;
```

In C:

```
pascal OSErr InitDateCache(DateCachePtr theCache);
```

This routine **must** be called before using the `String2Date` or `String2Time` routines to format the `theCache` record. Allocation of this record is the responsibility of the caller: it can either be a local variable, a `Ptr` or a locked `Handle`. By using this cache, the performance of the `String2Date` and `String2Time` routines is improved.

In Pascal:

```
Procedure MyRoutine;
        Var
                myCache: DateCacheRecord;
        Begin
                InitDateCache (@myCache);
                {call the String2Date or Time routines.  Note that if you are doing this}
                {inside an application where global variables are allowed, you should probably}
                {make your Date cache a global and initialize it once, when you initialize}
                {the Toolbox Managers.}
        End;
```

In C:

```
void MyRoutine()
{
        DateCacheRecord       myCache;

        InitDateCache(&myCache);
        /* Now you can call String2Date or String2Time, Note that if you are doing this
        inside an application where global variables are allowed, you should probably make
        your Date cache a global and initialize it once when you initialize the Toolbox
        managers
        */
}
```

String2Date and String2Time

In Pascal:

```
Function String2Date(textPtr:       Ptr;
                     textLen:       longint;
                     theCache:      DateCachePtr;
                Var  lengthUsed:    Longint;
                Var  dateTime:      LongDateRec)
        : String2DateStatus;

Function String2Time(textPtr:       Ptr;
                     textLen:       longint;
                     theCache:      DateCachePtr;
                Var  lengthUsed:    Longint;
                Var  dateTime:      LongDateRec)
        : String2DateStatus;
```

In C:

```
pascal String2DateStatus String2Date(Ptr textPtr,long textLen,DateCachePtr theCache,
    long *lengthUsed,LongDateRec *dateTime);

pascal String2DateStatus String2Time(Ptr textPtr,long textLen,DateCachePtr theCache,
    long *lengthUsed,LongDateRec *dateTime);
```

These routines expect a date and time at the beginning of the text. They parse the text, setting the `lengthUsed` to reflect the remainder of the text, and fill the `dateTime` record. They recognize all the strings that are produced by the international date and time utilities, and others. For example, they will recognize the following dates: September 1, 1987; 1 Sept 1987; 1/9/1987; and 1 1987 sEpT.

If the value of the input year is less than 100, then it is added to 1900; if less than 1000, then it is added to 1000 (the appropriate values are used from other calendars, gotten from the base date: LongDateTime = 0). Thus the dates 1/9/1987 and 1/9/87 are equivalent.

The routines use the following grammar to interpret the date and time. The relevant fields of the international utilities resources are used for separators, month and weekday names, and the ordering of the date elements. The parsing is actually semantic-driven, so finer distinctions are made than those represented in the syntax diagram.

```
time   := number [tSep number [tSep number]] [mornStr | eveStr | timeSuff]
tSep   := timeSep | sep
date   := [dSep] dField [dSep dField [dSep dField [dSep dField [dSep]]]]
dField := number | dayOfWeek | abbrevMonth | month
dSep   := dateSep | st0 | st1 | st2 | st3 | st4 | sep
sep    := <non-alphanumeric>
```

The date defaults are the current day, month and year. The time defaults to 00:00:00. The digits in a year are padded on the left, using the base date (the date corresponding to zero seconds: Jan 1, 1904). This routine uses the tokenizer to separate the components of the strings. It depends upon the names of the months and weekdays used from international resources being single alphanumeric tokens.

Note that the date routine only fills in the year, month, day and dayOfWeek; the time routine fills in only the hour, minute and second. Thus the two routines can be called sequentially to fill complementary values in the LongDateRec.

The return from the routine is a set of bits that indicate confidence levels, with higher numbers indicating low confidence in how closely the input string matched what the routine expected. For example, inputting a time of 12.43.36 will work, but return a message indicating that the separator was not standard. This can also be used to parse a string containing both the date and time, by using the confidence levels to determine which portion comes first. The returned bits include:

In Pascal:

```
fatalDateTime = $8000;
longDateFound = 1;
leftOverChars = 2;
sepNotIntlSep = 4;
fieldOrderNotIntl = 8;
extraneousStrings = 16;
tooManySeps = 32;
sepNotConsistent = 64;
tokenErr = $8100;
cantReadUtilities = $8200;
dateTimeNotFound = $8400;
dateTimeInvalid = $8800;
```

In C:

```
#define fatalDateTime 0x8000
#define longDateFound 1
#define leftOverChars 2
#define sepNotIntlSep 4
#define fieldOrderNotIntl 8
#define extraneousStrings 16
#define tooManySeps 32
#define sepNotConsistent 64
#define tokenErr 0x8100
#define cantReadUtilities 0x8200
#define dateTimeNotFound 0x8400
#define dateTimeInvalid 0x8800
```

LongDate Conversion

In Pascal:

```
Procedure LongDate2Secs(      lDate: LongDateRec;
                      Var     lSecs: LongDateTime);

Procedure LongSecs2Date(      lSecs: LongDateTime;
                      Var     lDate: LongDateRec);
```

In C:

```
pascal void LongDate2Secs(const LongDateRec *lDate,LongDateTime *lSecs);

pascal void LongSecs2Date(LongDateTime *lSecs,LongDateRec *lDate);
```

These routines extend the range of the Macintosh calendar as discussed above. Any fields that are not used should be zeroed. On input, the `LongDate2Secs` routine will use the day and month unless the day is zero; otherwise the dayOfYear is used unless it is zero; otherwise the dayOfWeek and weekOfYear are used.

Other fields are additive: if you supply a month of 37, that will be interpreted as adding 3 to the year, and using a month of 1. This latter property is subject to some restrictions imposed by the internal arithmetic: for example, | hour*60+minute | must be less than 32767.

Two new interfaces have been added to Pack6 for `LongDate` support:

In Pascal:

```
IULDateString(        dateTime:       LongDateTime;
                      form:           DateForm;
              Var     Result:         Str255;
                      intlParam:      Handle);
Assembly selector:    20

IULTimeString(        dateTime:       LongDateTime;
                      wantSeconds:    BOOLEAN;
              Var     Result:         Str255;
                      intlParam:      Handle);
Assembly selector:    22
```

In C:

```
pascal void IULDateString(LongDateTime *dateTime,DateForm longFlag,Str255 result,
    Handle intlParam);

pascal void IULTimeString(LongDateTime *dateTime,Boolean wantSeconds,Str255 result,
    Handle intlParam);
```

These routines take a LongDateTime, and return a formatted string. Only the old fields year..second, and dayOfWeek are used. If the intlParam is zero, then the international resource 0 ('itl0') is used. The output year is limited to four digits: e.g., from 1 to 9999 A.D.

ToggleDate and ValidDate

In Pascal:

```
Function ToggleDate (Var     mySecs:        LongDateTime;
                             field:         LongDateField;
                             delta:         DateDelta;
                             ch:            Integer;
                             params:        TogglePB)
       :ToggleResults;

Function ValidDate (  Var    date :         LongDateRec;
                             flags:         Longint;
                      Var    newSecs:       LongDateTime)
       : Integer;
```

In C:

```
pascal ToggleResults ToggleDate(LongDateTime *lSecs,LongDateField field,
    DateDelta delta,short ch,const TogglePB *params);

pascal short ValidDate(LongDateRec *vDate,long flags,LongDateTime *newSecs);
```

The ToggleDate routine is used to modify a date or time record by toggling one of the fields up or down. The routine returns a valid date by performing two types of action. If the affected field overflows or underflows, then it will wrap to the corresponding low or high value. If changing the affected field causes other fields to be invalid, then a close date is selected (which may cause other fields to change). For example, toggling the year upwards in February 29, 1980 results in March 1, 1981. Currently only the fields year..second, and am can be toggled, although this should change in the future.

The routine will also toggle by character, if the delta = 0. The character will be used to change the field in the following way. If it is a digit, then it will be added to the end of the field, and the field will be then modified to be valid in a similar manner as in the alarm clock. For example, if the minute is 54, then to replace it by 23 by entering characters, first the minute will change to 42, then to 23. The AM/PM field will also use letters.

In Pascal:

```
TogglePB = RECORD
    togFlags: LONGINT;
    amChars: ResType;                              {from intl0}
    pmChars: ResType;                              {from intl0}
    reserved: ARRAY [0..3] OF LONGINT;
    END;
```

In C:

```
struct TogglePB {
    long togFlags;
    ResType amChars;                /*from intl0*/
    ResType pmChars;                /*from intl0*/
    long reserved[4];
};
```

The parameter block should be set up as follows. It should contain the uppercase versions of the AM and PM strings to match (the defaults mornStr and eveStr can be copied from the international utilities using IUGetIntl, and converted to uppercase with UprText).

The ToggleDate routine makes an internal call to ValidDate, which can also be called directly by the user. ValidDate checks the date record for correctness, using the params.togflags which is passed to it by ToggleDate. If any of the record fields are invalid, ValidDate returns a DateField value corresponding to the field in error. Otherwise, it returns a -1.

The params.togflags value passed to ValidDate by ToggleDate are the same for ToggleDate and ValidDate. The low word bits correspond to the values in the enumerated type DateField. For example, to check the validity of the year field you can create a mask by doing the following:

```
yearFieldMask = 2**yearField;
```

The high word of the flags value can be used to set various other conditions. The only one currently used is a flag which can be set to restrict the range of valid dates to the short date format (smallDateBit = 31; smallDateMask = $80000000). All other bits are reserved, and should be set to zero. The reserved values should also be zeroed.

Togflags should normally be set to $007F, which can be done by using the predeclared constant dateStdMask.

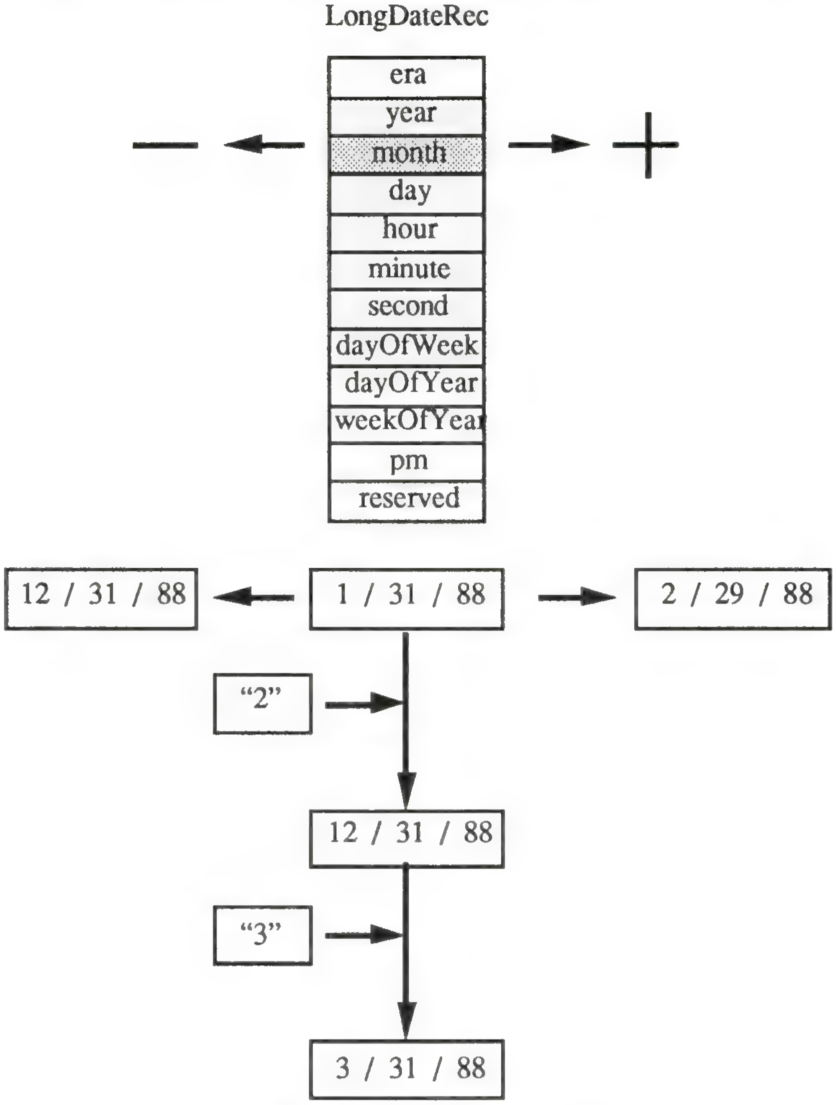


Figure 8–ToggleDate

Reading and Writing the Location

In Pascal:

```
PROCEDURE ReadLocation(VAR loc: MachineLocation);
PROCEDURE WriteLocation(loc: MachineLocation);
```

In C:

```
pascal void ReadLocation(MachineLocation *loc);
pascal void WriteLocation(const MachineLocation *loc);
```

These routines allow the programmer to access the stored geographic location of the Macintosh and time zone information from parameter RAM. For example, the time zone information can be used to derive the absolute time (GMT) that a document or mail message was created. With this information, when the document is received across time zones, the creation date and time are correct. Otherwise, documents can appear to be created after" they are read (e.g., I can create a message in Tokyo on Tuesday and send it to Cupertino, where it is received and read on Monday). Geographic information can also be used by applications which require it.

If the MachineLocation has never been set, then it should be <0,0,0>. The top byte of the gmtDelta should be masked off and preserved when writing: it is reserved for future extension. The gmtDelta is in seconds east of GMT: e.g., San Francisco is at minus 28,800 seconds (8 hours * 3600 seconds per hour). The latitude and longitude are in fractions of a great circle, giving them accuracy to within less than a foot, which should be sufficient for most purposes. For example, Fract values of 1.0 = 90°, -1.0 = -90°, -2.0 = -180°.

In Pascal:

```
MachineLocation = RECORD
    latitude: Fract;
    longitude: Fract;
    CASE INTEGER OF
      0:
        (dlsDelta: SignedByte);                {signed byte; daylight savings delta}
      1:
        (gmtDelta: LONGINT);                   {must mask - see documentation}
    END;
```

In C:

```
struct MachineLocation {
    Fract latitude;
    Fract longitude;
    union{
        char dlsDelta;                /*signed byte; daylight savings delta*/
        long gmtDelta;                /*must mask - see documentation*/
        }gmtFlags;
};
```

The gmtDelta is really a three-byte value, so the user must take care to get and set it properly as in the following code examples:

In Pascal:

```
Function GetGmtDelta(myLocation: MachineLocation): longint;
Var
       internalGmtDelta: Longint;
begin
       With myLocation Do Begin
              internalGmtDelta := BAnd(gmtDelta,$00FFFFFF);      {get value}
              If BTst(internalGmtDelta,23)                       {sign extend}
                     Then internalGmtDelta := BOr(internalGmtDelta,$FF000000);
       GetGmtDelta := internalGmtDelta;
       End;
End;

Procedure SetGmtDelta(Var myLocation: Location; myGmtDelta: Longint);
Var
       tempSignedByte: SignedByte;

BEGIN
       WITH myLocation DO BEGIN
              tempSignedByte := dlsDelta;
              gmtDelta := myGmtDelta;
              dlsDelta := tempSignedByte;
       END;
END;
```

In C:

```
long GetGmtDelta(MachineLocation myLocation)
{
        long    internalGMTDelta;

        internalGMTDelta = myLocation.gmtDelta & 0x00ffffff;

        if ( (internalGMTDelta >> 23) & 1 ) // need to sign extend
                internalGmtDelta = internalGmtDelta | 0xff000000;

        return(internalGmtDelta);
}

void SetGmtDelta(MachineLocation *myLocation, long myGmtDelta)
{
        char tempSignedByte;

        tempSignedByte = myLocation->dlsDelta;
        myLocation->gmtDelta = myGmtDelta;
        myLocation->dlsDelta = tempSignedByte;
}
```

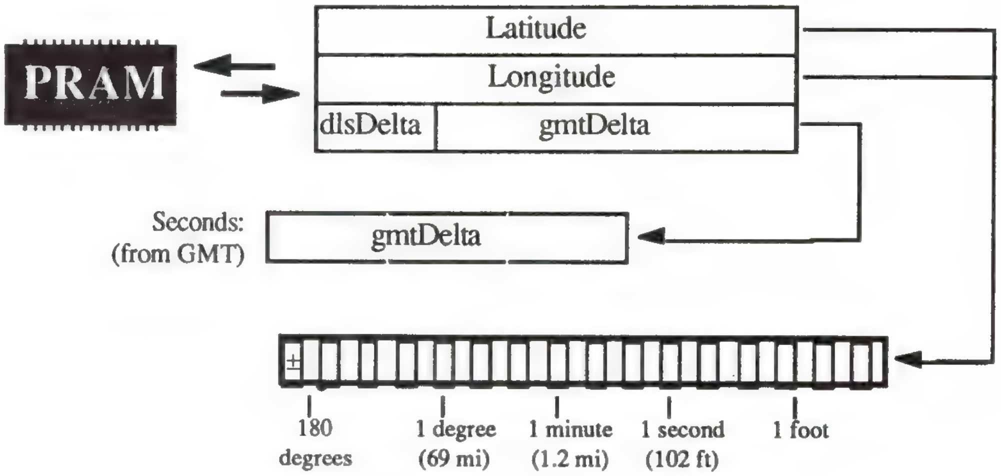
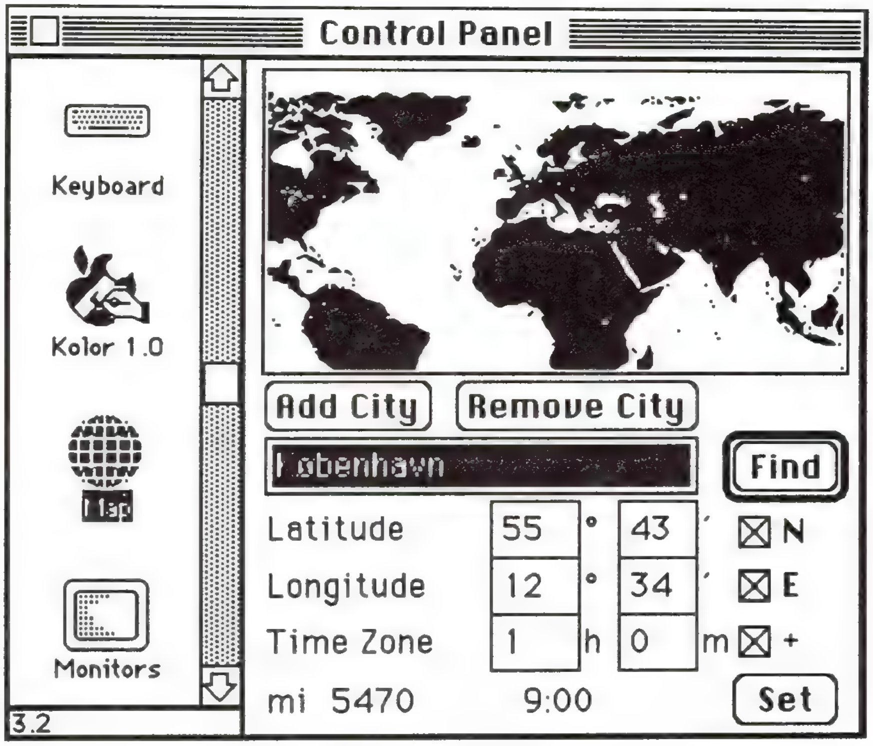


Figure 9–Locations

Setting Latitude, Longitude, and Time Zone cdev

This new Control Panel module on the utilities disk allows the user to set the latitude, longitude, and time zone. The values are stored in parameter RAM on the host machine. (See the Map cdev documentation for more details).



Kobenhavn

Latitude 55 ° 43 ' ☒ N
Longitude 12 ° 34 ' ☒ E
Time Zone 1 h 0 m ☒ +
mi 5470 9:00

Figure 10–Map

Numbers

The new number routines supplement SANE, allowing applications to display formatted numbers in the manner of Microsoft Excel or Fourth Dimension, and to **read** both formatted and simple numbers. The formatting strings allow natural display and entry of numbers and editing of format strings even though the original numbers **and** the format strings were entered in a language other than that of the final user.

Number parsing is based on a `NumberParts` table that describes the essentials of numeric display for a particular language, including such components as thousands separator, decimal point, scientific notation, forced zeroes in the absence of significant digits, etc. A default `NumberParts` table for each locale's system resides in the `'itl4'` resource for that system.

In Pascal:

```
NumberParts = RECORD
        version:                        integer;
        data:                           array [tokLeftQuote..tokMaxSymbols] OF WideChar;
        pePlus, peMinus, peMinusPlus:   WideCharArr;
        altNumTable:                    WideCharArr;
        reserved:                       packed array [0..19] of Char; {must be zeroed!}
END;
```

In C:

```
struct NumberParts {
    short version;
    WideChar data[31];                     /*index by [tokLeftQuote..tokMaxSymbols]*/
    WideCharArr pePlus;
    WideCharArr peMinus;
    WideCharArr peMinusPlus;
    WideCharArr altNumTable;
    char reserved[20];
};
```

Here is an example of how to access the `'itl4'` default `NumberParts` table:

In Pascal:

```
Function DefaultParts( Var x: NumberParts ): Boolean;
Var
        itl4: Itl4Handle;
Begin
        DefaultParts := false;                                   {assume error}
        itl4 := itl4Handle(IUGetIntl(4));                        {get itl4 record}
        if itl4 <> nil then begin                                {if ok}
                x := NumberPartsPtr(ord(itl4^)+itl4^^.defPartsOffset)^;
                        {number parts subtable}
                DefaultParts := true;                            {no error}
        end;
End;
```

In C:

```
DefaultParts(NumberParts *x)
{
        Itl4Handle itl4;

        itl4 = (Itl4Handle)IUGetIntl(4);

        if ( itl4 ) {
                *x = *((NumberPartsPtr)( (char *)(*itl4) + ((*itl4)->defPartsOffset ) ) );
                return(1);
        }

        return(0);
}
```

The user provides a format descriptor string very similar to Fourth Dimension's. This format string is translated by `Str2Format` in a canonical format which is transportable between different languages such as French, English, and Japanese. The canonical format is stored in a record called `NumFormatString`. This record's structure is as follows:

In Pascal:

```
NumFormatString = PACKED RECORD
    fLength: Byte;
    fVersion: Byte;
    data: PACKED ARRAY [0..253] OF SignedByte;            {private data}
    END;
```

In C:

```
struct NumFormatString {
    char fLength;
    char fVersion;
    char data[254];                    /*private data*/
};
```

The format descriptor string may be broken into as many as three parts: positive, negative, and zero. For example, the number 3456.713 used with the canonical format produced from "#,###.#;(#,###.#)" will produce the string representation "3,456.7" in the United States. In Switzerland the same canonical format would be displayed as "#.###,#;(#.###,#)," and the number displayed with this format would be "3.456,7."

The number formats include the following features (the defaults for the U.S. are listed following):

Separators:

decimal separator (.), thousands separator (,)

Example:	format string: ###,###.0##,###	
1	—>	1.0
1234	—>	1,234.0
3.141592	—>	3.141,592

Digits:

zero digit (0), skipping digit (#), padding digit (^), padding value (NBSP)

Example:	format string: ###;(000);^^^	
1	—>	1
-1	—>	(001)
0	—>	0

The number format routines always fill in digits from the right or from the left of the decimal point.

Example:	format string: ###'foo'###	
123foo456	—>	123foo456
22foo44	—>	2foo244
123foo	—>	123

Example:	format string: 0.###'foo'###	
0.foo123	—>	0.123
0.1foo456	—>	0.145foo6
0.1456	—>	0.145foo6

Formats using zero and skipping digit characters do not allow extension beyond the minimum number of digits specified to the right or left of the decimal place. For example: users must provide the desired maximum digits on the left: e.g., #,###,### instead of #,###. `X2FormStr` will return a result of `formatOverflow` when the number contains more digits to the left of the decimal point than specified in the format string. Input values with more digits to the right of the decimal point than there are digits allowed in the format string will be rounded on output.

Example:	format string: ###.###	
1234.56789	—>	formatOverflow on output
1.234999	—>	1.235

Control:

left quote (‘), right quote (’), escape quote (\), sign separator (;)

Example:	format string: ###‘CR’;###‘DB’;‘\’zero\‘’	
1	—>	1CR
-1	—>	1DB
0	—>	‘zero’

Marks:

plus (+), minus (-), percent (%), positive exponent (E+), negative exponent (E-), mixed exponent (E)

Example:	format string: ##%	
0.1	—>	10%

There is a limitation creating format strings with exponential notation: the user must always place zero leaders immediately after the exponent marks and skipping digits before, when more than one digit must be represented between the exponent and the decimal point.

Example:	format string: ##.####E+0	
1.23E+3	—>	1.23E+3

The sign of exponents must be made explicit in the format string by using ePlus (E+) or eMinus (E-) format. eMinusPlus notation (E) is only used in the input number string to specify a positive exponent when the sign of the format string exponent is negative.

	exponent sign	
format	+	-
ePlus	ePlus(E+)	eMinus(E-)
eMinus	eMinusPlus(E)	eMinus(E-)

Use ePlus notation in the format string to specify negatively or positively signed exponents in the input number string:

Example:	ePlus format string: #.#E+#		
1.2E-3	—>	1.2E-3	
1.2E+3	—>	1.2E+3	

Example:	eMinus format string: #.#E-#		
1.2E-3	—>	1.2E-3	
1.2E3	—>	1.2E3	(i.e., 1200)

Literals:

unquoted literals ([]$:(){}), literals requiring quotes (ABC...)

Example:	format string: [###‘ Million ’###‘ Thousand ’###]	
300	—>	[300]
3000000	—>	[3 Million 000 Thousand 000]

A typical scenario consists of the application reading the default `NumberParts` table from `'itl4'`. One provides a format definition string, such as the string "#.###,#;(#.###,#)" of the above example, as a template for whatever field one is currently working in. The application submits that string to `Str2Format`, which returns a canonical format string corresponding to the user's input. This canonical format, rather than the raw format definition string, is stored in the document. The program can convert the canonical format back to a user-editable string using the `Format2Str` routine.

When a number is to be displayed, the application passes the number and canonical format to `FormatX2Str` to produce a formatted number that the application then displays in that field. If the user types a string into the field, then `FormatStr2X` can be used with the canonical format for the field to read formatted numbers. That is, the user can type "(3.678,9)" and have the number interpreted correctly.

Converting to Canonical Formats

In Pascal:

```
FUNCTION Str2Format(inString: Str255;partsTable: NumberParts;
                    VAR outString: NumFormatString): FormatStatus;
```

In C:

```
pascal FormatStatus Str2Format(const Str255 inString,const NumberParts *partsTable,
    NumFormatString *outString);
```

`Str2Format` converts a string typed by the user into a canonical format. It checks the validity of the format string itself and also that of the `NumberParts` table, because the `NumberParts` table is programmable by the application.

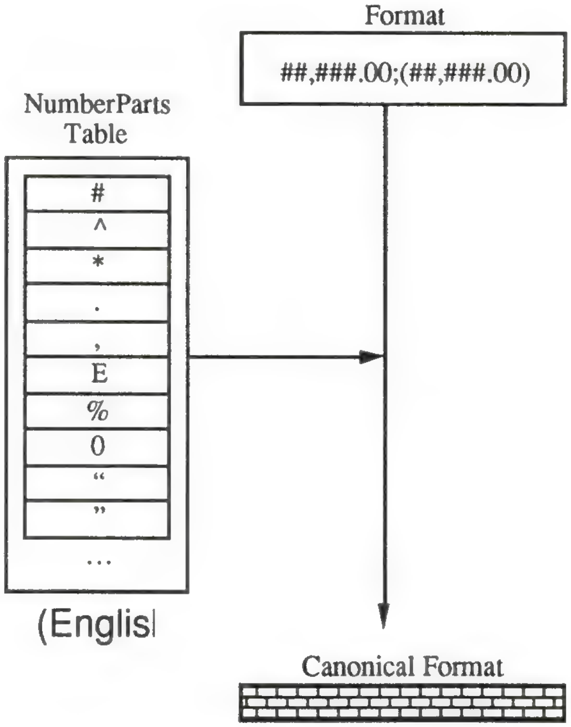


Figure 11–Str2Format

Displaying the Canonical Format String

In Pascal:

```
FUNCTION Format2Str(myCanonical: NumFormatString;partsTable: NumberParts;
    VAR outString: Str255;VAR positions: TripleInt): FormatStatus;
```

In C:

```
pascal FormatStatus Format2Str(const NumFormatString *myCanonical,
const NumberParts *partsTable,Str255 outString,TripleInt *positions);
```

`Format2Str` creates the string corresponding to a format definition string which has been created by a prior call to `Str2Format` and according to the `NumberParts` table. It is the inverse operation of `Str2Format`. This allows programs to display previously entered formats for users to edit.

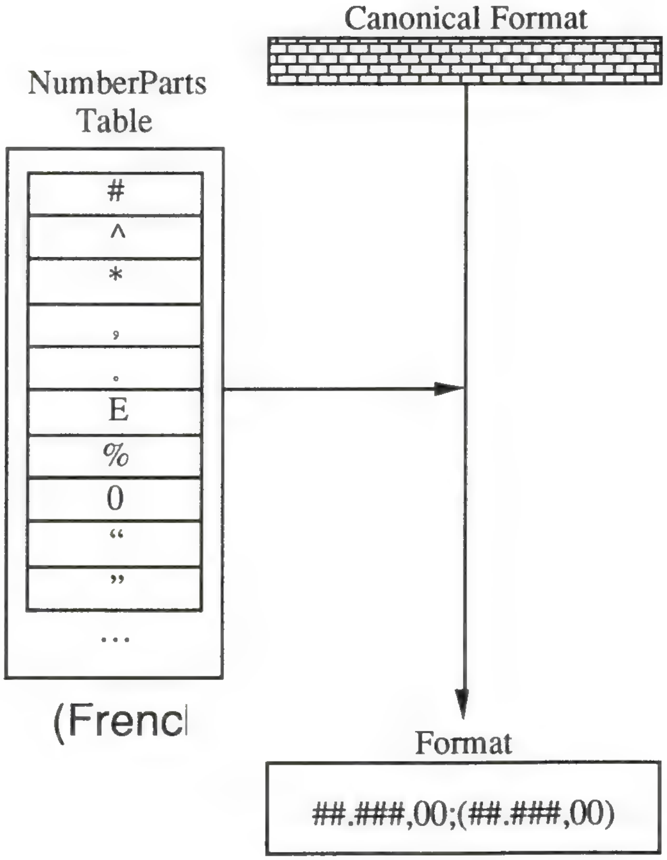


Figure 12–Format2Str

Formatting Numbers

In Pascal:

```
FUNCTION FormatX2Str(x: Extended;myCanonical: NumFormatString;partsTable: NumberParts;
    VAR outString: Str255): FormatStatus;
```

In C:

```
pascal FormatStatus FormatX2Str(extended x,const NumFormatString *myCanonical,
    const NumberParts *partsTable,Str255 outString);
```

This routine creates a textual representation of a number according to a canonical format which has been created by a prior call to `Str2Format`.

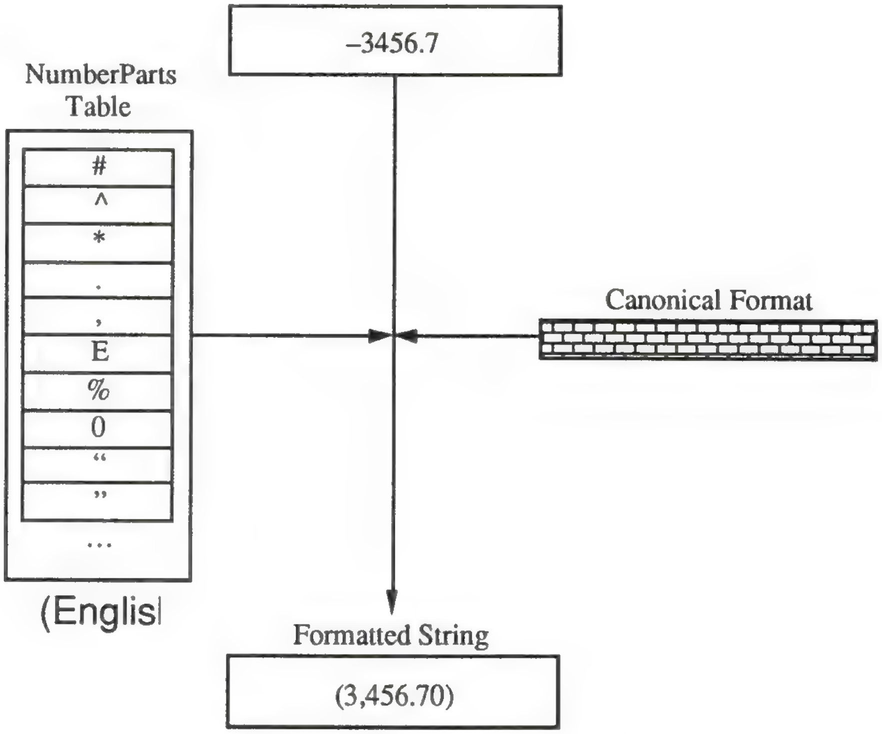


Figure 13–FormatX2Str

Reading Formatted Numbers

In Pascal:

```
FUNCTION FormatStr2X(source: Str255;myCanonical: NumFormatString;partsTable: NumberParts;
    VAR x: Extended): FormatStatus;
```

In C:

```
pascal FormatStatus FormatStr2X(const Str255 source,const NumFormatString *myCanonical,
    const NumberParts *partsTable,extended *x);
```

This routine reads a textual representation of a number according to a canonical format which has been created by a prior call to `Str2Format`, and creates an extended floating point number which corresponds to that string.

Internally, the routine converts the string into a format acceptable to SANE, matching against the three possible patterns in the canonical format. If the input string does not match any of the patterns, then `FormatStr2X` parses the string as best it can returning the result. Currently it is converted to a simple form, stripping non-digits and replacing the decimal point, before calling SANE.

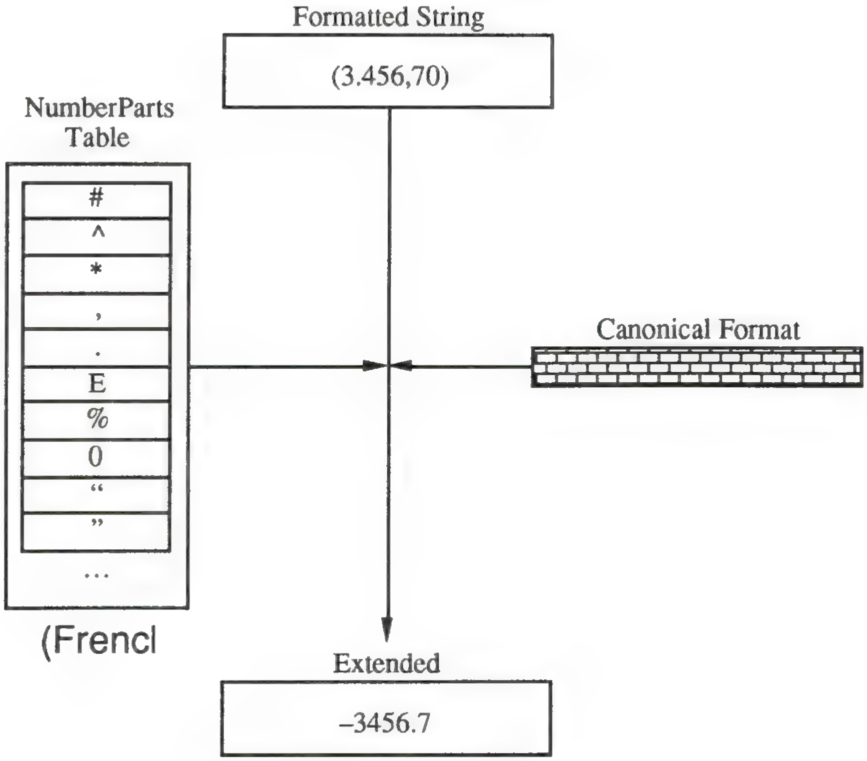


Figure 14–FormatStr2X

Summary of Routines

The updated MPW 3.0 interface files for the Script Manager 2.0 routines are available on AppleLink in the Developer Services Bulletin Board (Developer Services:Developer Technical Support:Macintosh:Latest MPW Interfaces) and from APDA as part of the MPW 3.0 final product.

Further Reference:

- *Inside Macintosh*, Volume V-293, The Script Manager
- Script Manager Hints and Recommendations (APDA)

Inside Macintosh Interim Chapter Draft

Translating Between Character Sets

Written by:	Karl Young	May 25, 1987
Last Revision:	Mark Davis & Karl Young	July 8, 1987

Synopsis

This document describes a method for converting (or, more correctly, transliterating) between different character sets. It defines a standard interface for calling a procedure for the transliteration. It describes the way procedures are stored as resources and how to describe a family of such procedures.

Motivation

Two current Macintosh applications have shown need for transliterating characters between different character sets, and there are many more.

Apple File Exchange transliterates files between IBM PC compatibles, the Apple II family, and the Macintosh. Each has a different character set which, especially in the case of the PC and Apple II, can vary from language to language and country to country.

Any telecommunications program that emulates terminals must deal with the varying character sets of each terminal. Once again, character sets on terminals vary on domestic and foreign versions.

It is difficult (and inelegant) to place the knowledge of all possible character sets into the code of the application. This document proposes a character conversion architecture using transliteration definition procedures for converting between different kinds of character sets. The main goals are to provide (a) a consistent interface for definition procedures, and (b) an extendable structure that allows new character sets to be handled by installation of new definition procedures.

Character Sets

We often describe the character set in terms of a machine that uses that character set. For instance, while the Macintosh, Apple II and IBM PC all use ASCII for the basis of their characters sets, they all have different extensions to ASCII which have different repertoires, and are coded differently. Each of these different encodings forms a different character set.

Similar character sets may also be slightly different in different countries. For example, the French version of EBCDIC is different from the English version. The German variant of ASCII substitutes umlauted characters ("ä","ö","ü") for ASCII punctuation ("[","/","]"). Each of these minor variations also form a different character set. These variant character sets are grouped as a *character set family*.

In general, characters sets may be composed of a number of subsets. For example, Xerox or ISO character sets are *shifting character sets*, allowing shifting between different character subsets by means of escape codes. On other machines, such as the Macintosh, the character set information is encoded by means of the font family number. The character conversion architecture treats different subsets as different character sets belonging to a character set family.

Character set families are distinguished by numbers, ranging from 0 to 32767. They are referenced below as being defined by:

```
        Type    CharFam = Integer;
```

An initial list of CharFam values consists of the following:

```
        cfMacintosh     =       0;
        cfASCII         =       1;
        cfIBMPC         =       2;
        cfIBMPS2        =       3;
        cfEBCDIC        =       4;
        cfISO8859       =       5;
        cfISO6937       =       6;
        cfXerox         =       7;
```

Countries are also numbered by country code, as initially defined in the International Utilities Package chapter of Inside Macintosh. Further country codes are defined in MPW equate files. Note that the term really refer to regions rather than countries: a country (Canada, Belgium, Switzerland) may contain a number of regions, each with its own country code. Country codes are defined as:

```
        Type    CntryCode = Integer;
```

Transliteration Procedure

Transliteration procedures can reside in application files, common resource files or in the system file. There may be many transliteration procedures (or TProc's for short) available in a particular resource file. Each TProc handles the transliteration from a particular source character set to a particular destination character set.

The call to the transliteration procedure can be described in Pascal as follows:

```
        Function Translit(    params:        TransPB): OSErr;


        Type    TransText = record
                              trPtr:         Ptr;
                              trLen:         Longint;
                              trFont:        Integer;
                        end;

                TransPB = record
                              verb:          Integer;       {i}
                              featureFlags:  Integer;       {i}
                              result:        OSErr;         {o}
                              newCountry:    Integer;       {io}
                              srcText:       TransText;     {io,io,i}
                              dstText:       TransText;     {io,io,io}
                              reserved:      Array [0..3] of Longint;
                        end;
```

result:

The function returns an error code (type OSERR) describing any error conditions that occurred during transliteration. This result is duplicated in the result field. In addition to the standard OSErr codes we define the following termination results for TProc's:

```
tDstTooShort   =      -501;  { destination buffer too short }
tSubsetSwitch  =      -502;  { source changed character subset }
tPartialChar   =      -503;  { source ends with partial character }
```

verb:

The verb can have the following values:

```
transInit              =      0;
transDone              =      1;
transLowToHigh         =      2;
transHighToLow         =      3;
```

Any other values are ignored (in current versions). The transInit and transDone verbs are called initially and finally for initialization and cleanup. Each transliteration procedure translates between two character sets. The transLowToHigh call translates from lower numbered character set to the larger number character set (e.g. Mac to IBM PC). The transHighToLow call translates in the reverse direction.

featureFlags:

The feature flags word contains a number of bits for selecting different variations of transliteration:

```
trMultiDestFont =     1;      {allow multiple destination fonts}
trMultiDestCountry =  2;      {allow multiple destination countries}
trNonOneToOne =       4;      {allow one:many or many:one conversions}
trOverStrike =        8;      {allow overstrike output, e.g. A-Backspace-`}
```

newCountry:

When a TProc is first used, this parameter should be set to -1 on input. On output, it reflects the country code that should be used for the next conversion (see below). If trMultiDestCountry is 0, this will always be the same as the current country. This parameter is used internally to insert control codes for shifting character sets.

srcText.trPtr, dstText.trPtr:

On input, these parameters are pointers to the source and destination buffers, respectively.

srcText.trLen, dstText.trLen:

As input, these parameters contain the size (in bytes) of the source and destination buffers, respectively. As output, srcText.trLen contains the number of bytes of the source string remaining that were not used; dstText.trLen contains the number of bytes remaining in the destination buffer.

Note that the number of bytes converted in the output text may differ from the number of bytes used in the input text for a number of reasons:

1. The destination may not be large enough for the required conversion; note that conversions may convert many bytes to one or vice versa unless trNonOneToOne is 0.
2. The input text may contain a partial character: the last byte may be part of a multibyte character.

3. To represent the input text on the specified machine, the font may need to be changed. For example, a symbol may be available in the source that is only available in the Symbol font on the Macintosh.

srcText.trFont, dstText.trFont:

These parameters contain the source and destination font numbers, respectively. For conversion to the Macintosh, these numbers are standard Font Manager font family numbers. For other machines distinguishing character sets by fonts, the font number may be different. A value of zero should refer to the standard (or system) font, while a value of one should refer to the common application font (which may be the same as the system font). On output, the dstText.trFont may contain a different font number, indicating that the proper representation of the text requires that font or an equivalent (e.g. Symbol). If the trMultiDestFont flag is 0, the output font will never differ from the input font: fallback characters will be used instead.

reserved:

This space should be zeroed on input.

Using the Transliteration Procedure

To use a transliteration procedure, the application should get the resource, lock it down for the duration, and jump to the start of the handle contents. (In Pascal, this can be done by recasting using a procedure pointer.) The srcSubset and dstSubset should be set to the initial character subset or font number (zero if unknown). The user must unlock the resource when done (after making the transDone call).

The transliterate routine will translate characters into the destination country and font (say, Times) as long as it can. If it is forced to stop because the source has changed to a font or country that it cannot handle, then it will indicate this by setting the destination country and font, and returning the error `subsetSwitch`. If the country changes, then the application should switch to a transliteration procedure that can handle that country (see below).

TProcs that handle shifting character sets should return the lowest-numbered country code for a TProc that will handle the new text.

As a general rule, control characters are not transliterated. Exceptions are in cases where characters are composed (e.g. in ISO 6937 where À equals *A-Backspace-`*), or where characters are represented by shifting to another graphic character set on one machine, but are single character on another.

Transliteration Procedure Resources

Transliteration procedures are stored as resources with type 'tprc'. The resources have a header of the form (in pseudo-Pascal):

```
Record
       {standard header}
branch:        Integer;      {BSR.S Code instruction}
flags: Integer;       {flag bits}
rType: ResType;       {'tprc'}
id:    Integer;       {resource id number}
version:       Integer;      {version number}
{tProc information}
lowCountry:    CntryCode;    {Macintosh country code}
```

```
lowCharFam:    CharFam;       {E.g. IBM PC Character set, French EBCDIC}
highCountry:   CntryCode;     {Macintosh country code}
highCharFam:   CharFam;       {E.g. Macintosh Character set}
reserved:      Array [0..3] of Longint;
End;
```

The resource ID numbers are assigned serially by Technical Support. The name of each resource should indicate the kind of transliteration that will be going on: e.g., “Mac to VT100,” “Mac to Apple IIe,” “EBCDIC to Mac,” etc. The country names can be gotten from the TProc family (see below).

The flags bits indicate which of the featureFlags the TProc will support. The low country is always the smaller number of the two countries that are translated between; the lowCharFam is the CharFam associated with the lowCountry.

One anticipated use of the resource name will be to display a list to users (say, of a terminal emulation program) so they can choose the appropriate transliteration. Remember that the resource names can be localized, so a program can’t necessarily depend on a certain name being present.

Transliteration Procedure Families

A TProc family is described by another resource, with resource type 'tprf'. It provides a way of finding the appropriate TProc for the countries and character sets involved. Note that the TProcs contain the specification information in their headers, so the TProc families (except for their names) can be reconstructed from the available TProcs.

The resource ID of this resource corresponds to the country code, and lists all of the available TProcs that transliterate to and from that country. The name of the resource also corresponds to the appropriate country, in the language of the host machine. Since this name may also be localized, programs cannot depend upon the names being constant.

The form of a TProc family resource is described as follows:

```
TPRF = RECORD
        lastEntry:     Integer;
        reserved:      Array[0..31] of Integer;
        TProcs:        Array[0..lastEntry] of TPRCEntry;
END;
```

The **lastEntry** field can be used to find the number of TPRC entries described in this family. The **reserved** field is for future extensions. The **TProcs** field is an array with **lastEntry**+1 elements, each element of the form described below:

```
TPRCEntry = RECORD
        tprcID:        Integer;
        curCharFam:    CharFam;
        altCountry:    CntryCode;
        altCharFam:    CharFam;
end;
```

The **tprcID** field contains the resource ID of the 'tprc' resource which will transliterate between the given countries and character sets, where the current country is derived from the family resource id. The **curCharFam** field contains the character set family number associated with the family’s country.

The **altCountry** field contains the country code for the second country. As noted earlier, this will most often be the same as the current country, but it need not be. The **altCharFam** field contains the second character set family number. Each TProc may have two entries: one in the TProc family for the lowCountry, and one for the highCountry. It will have only one entry when the low and high countries are identical.

Using Transliteration Procedure Families

How an application uses a TProc family depends upon the amount of user interaction that can be required. An application can infer the current source and destination country from the international resource 0 (use IUGetIntl). Likewise, the user might want to choose from the list of available countries (which is a list of the names of the 'tprf' resources) for both the source and destination country.

For terminal emulation, either the source or the destination will always be the Macintosh, but the source and destination countries may vary.

Given a particular source and destination machine, the application can investigate the appropriate TProc family for the appropriate TProc. In the case of Apple File Exchange, where outside parties are writing file translators, the transliteration part of their code is abstracted so the translator will work across all countries without any further localization.

Human Interface Notes

Note #0 **Welcome to Human Interface Notes** **April 1990**

Human Interface Note #0 (this document) accompanies each release of Human Interface Notes. This release includes Human Interface Notes 7-8. These documents are written and produced by the Macintosh Human Interface Group and published in conjunction with Developer Technical Support for all Apple developers.

These documents correct and enhance the *Human Interface Guidelines: The Apple Desktop Interface*; they also incorporate and supersede the previously released "Human Interface Updates." This means that all you need is the *Human Interface Guidelines* and these documents to stay abreast of the latest recommended interface guidelines (unless, of course, you want to be on the very cutting edge of interface technology and read Tog's article in *Apple Direct* every month). These documents and the book will eventually be incorporated into a single, comprehensive human interface reference, but you'll need both until that time.

These documents are not done in a vacuum. Many of the guidelines you see are a direct result of developer feedback. If you have any questions or ideas for interface extensions or clarifications, please contact the Macintosh Human Interface Group at one of the following electronic addresses:

AppleLink: MacInterface
Internet: MacInterface@AppleLink.Apple.com

We want Human Interface Notes to be distributed as widely as possible, so they are sent to all Partners and Associates at no charge; they are also posted on AppleLink in the Developer Services bulletin board and other electronic sources, including the Apple FTP site (IP 130.43.2.2).

We place no restrictions on copying Human Interface Notes, with the exception that you cannot resell them, so read, enjoy, and share. We hope these guidelines provide you with lots of useful information while you are developing software for Apple computers. The following page lists all Human Interface Notes that have been released.

Released Human Interface Notes

April 1990

*** New
R Revised (Supersedes Human Interface Update)

	Number	Title	Released
	1	User Observation: Guidelines for Apple Developers	1/90
	2	Design Principles for On-Line Help Systems	1/90
	3	Dueling Metaphors: the Desktop & HyperCard	1/90
	4	Movable Modal Dialog Boxes	1/90
	5	What "Cancel" Means	1/90
	6	Window Positions	1/90
***	7	Who's Zooming Whom?	4/90
***	8	Keyboard Equivalents	4/90

Human Interface Notes

Note #1 **User Observation: Guidelines for Apple Developers**

Written by: Kathleen Gomoll & Anne Nicol
(Supersedes Human Interface Update #11)

January 1990

Discussion of guidelines for user observation.

Introduction

User testing covers a wide range of activities designed to obtain information on the interactions between users and computers. Most user testing requires considerable expertise in research methods, as well as skill in using complex data collection tools. For example, user testing techniques include: interviews, focus groups, surveys, timed performance tests, keystroke protocols, and controlled laboratory experiments. Of the many user testing techniques available, user observation is one technique that can be used by anyone with a concern for including the user in the product development process.

User observation involves watching and listening carefully to users as they work with a product. Although it is possible to collect far more elaborate data, observing users is a quick way to obtain an objective view of a product.

When to observe users

User observation should be an integral part of the design process—from the initial concept to the product's release. Software design that includes user observation is an iterative process; user feedback provides the data for making design modifications.

As Figure 1 demonstrates, this iterative process assumes that preliminary human interface designs should exist prior to the development of underlying code. Interface designs should be tested frequently to determine which design should be implemented. Then, as the code develops, the entire product should be tested and revised several times.

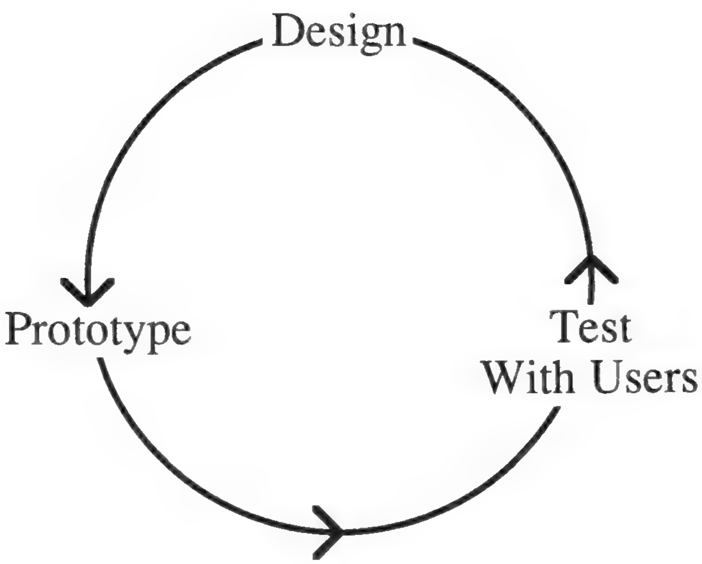


Figure 1–User observation in software design

Preparing for a user observation

Set an objective

Before you do any testing, you should take time to figure out what you're testing and what you're not. In other words, determine an objective for your test that focuses on a specific aspect of the product. By limiting the scope of the test, you're more likely to get information that helps you solve a specific problem.

Design the tasks

Your test participant will work through one or more specific tasks. These tasks should be real tasks that you expect most users will do when they use your product. The entire user observation should not run over an hour, so you should design tasks that focus on the part of the product you're studying. For example, if you want to know whether your menus are useful, you could design a task that requires the participant to access the menus frequently. After you determine which tasks to use, write them out as short, simple instructions.

> **Important:** Your instructions must be clear and complete, but they should not explain how to do things you're trying to test. For example, if you want to find out whether users can navigate through your program easily, don't give them instructions for navigation. Or, if you want to know whether your interface is self-explanatory, don't describe how it works. This concept is extremely important to remember. If you teach your participants about something you're trying to test, your data will not be useful.

Decide upon the use of videotape

Although you can observe users effectively without using special recording equipment, you may want to use videotape to capture the entire session. By videotaping the session, you collect an enormous amount of valuable information that you can review and analyze after the test is over. If video equipment is not available, a tape recorder can be helpful for recording what is said during the test.

Determine the setting

The ideal setting for user observation is a quiet, enclosed room with a desk, the appropriate hardware and software, a video camera, and two microphones (one for you and one for the participant). Of course, you may not have all these things available when you need to observe; therefore, you should try to approximate the ideal setting as closely as you can. If you have to conduct the observation in a regular office, ask the people around you to keep the noise level down during the observation. The key is to make the environment as interruption-free as possible. Get the participants out of their offices, away from phone calls and people who might drop by.

Find representative users

When looking for participants, try to find people who have the same experience level as the typical user for your product. Don't ask people you work with regularly to be participants because they are probably familiar with your product or your opinions about the product. Generally, you should look for people who are familiar with the hardware you use but are not familiar with your product.

You may want to ask pairs of people to work together on your tasks. You'll find that people working in pairs usually talk more than people working alone, and they also tend to discuss features of the product and explain things to each other.

10 steps for conducting a user observation

The following instructions guide you through a simple user observation. Remember, this test is not designed as an experiment, so you will not get statistical results. You can, however, see where people have difficulty using your product, and you can use that information to improve it.

These instructions are organized into steps. Under most of the steps, there is some explanatory text and a bulleted list. The bulleted list contains sample statements that you can read to the participant. (Feel free to modify the statements to suit your product and the situation.)

1. Introduce yourself.

2. Describe the purpose of the observation (in general terms).

Set the participant at ease by stressing that you're trying to find problems in the product. For example, you could say:

- You're helping us by trying out this product in its early stages.
- We're looking for places where the product may be difficult to use.
- If you have trouble with some of the tasks, it's the product's fault, not yours. Don't feel bad; that's exactly what we're looking for.
- If we can locate the trouble spots, then we can go back and improve the product.
- Remember, we're testing the product, not you.

3. Tell the participant that it's okay to quit at any time.

Never leave this step out. Make sure you inform participants that they can quit at any time if they find themselves becoming uncomfortable. Participants shouldn't feel like they're locked into completing tasks. Say something like this:

> "Although I don't know of any reason for this to happen, if you should become uncomfortable or find this test objectionable in any way, you are free to quit at any time."

4. Talk about the equipment in the room.

Explain the purpose of each piece of equipment (hardware, software, video camera, microphones, etc.) and how it is used in the test.

5. Explain how to "think aloud."

Ask participants to think aloud during the observation, saying what comes to mind as they work. By listening to participants think and plan, you can examine their expectations for your product, as well as their intentions and their problem solving strategies. You'll find that listening to users as they work provides you with an enormous amount of useful information that you can get no other way.

Unfortunately, most people feel awkward or self-conscious about thinking aloud. Explain why you want participants to think aloud, and demonstrate how to do it. For example, you could say:

- We have found that we get a great deal of information from these informal tests if we ask people to think aloud as they work through the exercises.
- It may be a bit awkward at first, but it's really very easy once you get used to it.
- All you have to do is speak your thoughts as you work.
- If you forget to think aloud, I'll remind you to keep talking.
- Would you like me to demonstrate?

6. Explain that you cannot provide help.

It is very important that you allow participants to work with your product without any interference or extra help. This is the best way to see how people really interact with the product. For example, if you see a participant begin to have difficulty and you immediately provide an answer, you lose the most valuable information you can gain from user observation—where users have trouble, and how they figure out what to do.

Of course, there may be situations where you have to step in and provide assistance, but you should decide what those situations might be before you begin testing. For example, you may decide that you can allow someone to flounder for at least 3 minutes before you provide assistance. Or, you may decide that there is a distinct set of problems with which you can provide help.

As a rule of thumb, try not to give your test participants any more information than the true users of your product will have. Following are some things you can say to the participant:

- As you're working through the exercises, I won't be able to provide help or answer questions. This is because we want to create the most realistic situation possible.
- Even though I won't be able to answer your questions, please ask them anyway. It's very important that I capture all your questions and comments on tape.
- When you've finished all the exercises, I'll answer any questions you still have.

7. Describe the tasks and introduce the product.

Explain what the participant should do and in what order. Give the participant written instructions for the tasks.

> **Important:** If you need to demonstrate your product before the user observation begins, be sure you don't demonstrate something you're trying to test. (For example, if you want to know whether users can figure out how to use certain tools, don't show them how to use the tools before the test.)

8. Ask if there are any questions before you start; then begin the observation.

9. Conclude the observation.

When the test is over:

- explain what you were trying to find out during the test.
- answer any remaining questions the participant may have.
- discuss any interesting behaviors you would like the participant to explain.

10. Use the results.

As you observe, you see users doing things you never expect them to do. When you see participants making mistakes, your first instinct may be to blame the mistakes on the participant's inexperience or lack of intelligence. This is the wrong focus. The purpose of observing users is to see what parts of your product might be difficult or ineffective. Therefore, if you see a participant struggling or making mistakes, you should attribute the difficulties to faulty product design, not to the participant.

To get the most out of your test results, review all your data carefully and thoroughly (notes, the video tape or cassette tape, the tasks, etc.). Look for places where participants had trouble, and see if you can determine how your product could be changed to alleviate the problems. Look for patterns in the participants' behavior that might tell you whether the product was understood correctly.

It's a good idea to keep a record of what you found during the test. That way, you have documentation to support your design decisions and you can see trends in users' behavior. After you've examined the results and summarized the important findings, fix the problems you found and test the product again. By testing your product more than once, you can see how your changes affect users' performance.

Human Interface Notes

Note #2 **Design Principles for On-Line Help Systems**

Written by: Kathleen Gomoll & Anne Nicol
(Supersedes Human Interface Update #12)

January 1990

Discussion of a set of criteria and guidelines for on-line help.

Introduction

As part of an ongoing effort to design and support a consistent interface to on-line help for our computers, Apple has been developing a set of criteria and guidelines for on-line help. These criteria are based on observations of users in our lab, reviews of the research, requests and comments from our developers, and—last, but not least—the *Human Interface Guidelines: The Apple Desktop Interface*. This is a working document. It reflects Apple's current view on designing on-line help, and we will probably revise and expand it as we progress. In the future, we intend to distribute specific guidelines regarding access to on-line help and the display of help information. Ultimately, we hope to supply toolbox support for the interface features that we find, through user testing, to be most effective.

This document is divided into three sections: Principles, General guidelines, and Hints for structure and organization. The Principles section reflects Apple's underlying design philosophy for on-line help. The General guidelines section puts guidelines for designing on-line help into the context of the principles outlined in *Human Interface Guidelines*. Finally, the Hints section lists suggestions for organizing and structuring help information. These hints come from developers and from current research.

Principles

On-line help should never be a substitute for good interface design.

This is our first and foremost principle. Before setting out to build a help system that "explains" a difficult interface, try to identify what makes the interface difficult—and fix the problems. When you have made your interface as clear as it can be, then develop a help system that aids users as they work.

Help should be context-sensitive; it should not take the user away from the task at hand.

Perhaps the biggest complaint users have about help systems is that they don't want to leave their current application to get help. When users are forced to leave the context of their problem, they often forget the specifics of the problem. Also, users often have trouble applying the help information once they get back to the application because the help is no longer visible.

Help systems should assist users in framing their questions and provide different types of help for different questions.

When users need help, they often turn to local experts and ask questions. Human experts are often able to help users frame their questions so they can get the appropriate answer. Once the right question has been asked, help can be delivered quickly. Users' questions fall into a relatively small number of distinct categories, and those categories call for different types of assistance. For example, we can make a clear distinction between the question "What is this?" and the question "How do I do this?"

Help systems should be dynamic and responsive to individuals.

Different users need different kinds of help because they have individual learning styles and needs. For example, some users may want to be shown exactly how to do something, while others may want to explore and learn by their mistakes. If possible, on-line help systems should make use of the user's competence, learning style, level of experience and past actions to provide appropriate help.

Users shouldn't need help on how to get help.

Help systems should be structurally simple and self-explanatory. Although your help system may require a few words of instruction (like "click here" or "select a topic"), don't fall into the trap of turning your help system into a complicated application that requires lengthy instructions.

General Guidelines

Make help accessible through recognition, not recall.

See-and-point (instead of remember-and-type): Users can choose any available action at any time—without having to remember a particular command or name. This paradigm requires only recognition, rather than recall, of the desired activities.

Put the help system under the user's control.

Direct manipulation: Users want to feel that they are in charge of the computer's activities. The user, not the computer, initiates and controls all actions. If the user attempts something risky, the computer provides a warning, but allows the action to proceed if the user confirms it. This approach "protects" the beginner but allows the user to remain in control.

Support exploratory behavior by making an interactive help system.

User Control: People learn best when they're actively engaged. Too often, however, the computer acts and the user merely reacts within a limited set of options. Allow users to try things out.

Place help options where they are visible to the user.

Direct manipulation: Users want topics of interest to be highlighted. They want to see what functions are available at any given moment.

See-and-point: Users rely on recognition, not recall; they shouldn't have to remember anything the computer already knows.

Use graphics, animation, and sound.

Principles of Graphic Communication: The real point of graphic design, which comprises both pictures and text, is clear communication. In the Apple Desktop Interface, everything the user sees and manipulates on the screen is graphic.

Metaphors from the real world: Whenever appropriate, use audio and visual effects that support a real-world metaphor. Use animation for modeling user actions. Use sound for orienting attention and reinforcing information.

Hints for Structure and Content

On-line help should not be simply an on-line version of the print documentation.

As a method for communication, computers provide opportunities that books can't provide. Use the computer's capacity to its fullest by designing a help system that brings help to the user rather than requiring the user to page through an on-line book. Use the computer to link information in useful ways, and to create graphics, sound, animation, and examples.

Organize the help system in very small, addressable chunks of information.

By creating a help system that is modular, you allow tremendous flexibility. Small chunks of information can be grouped in strategic ways to provide users with only the most relevant information.

Include both search and browse capabilities.

Build a "find" feature into your help system to allow users to quickly search for specific topics. Also allow users to browse through available help topics, since it's often easier to recognize a topic than to think of an appropriate keyword.

Allow users to discard help.

Users should never be forced to use the help system to use an application. A help system should always be an optional aid.

Make the help system customizable and editable.

To make the most efficient use of a help system, users should be able to customize and edit the information to suit their own needs. For example, users may want to put a marker on a piece of information they access frequently, or they may want to eliminate or change information that doesn't suit their needs.

Include help information that can be delivered automatically.

Sometimes users make the same error repeatedly. Rather than waiting for the user to ask for help, the help system should be able to detect problems and offer help automatically.

Incorporate hypertext features for linking information.

Hypertext allows users to press "buttons" to receive context-sensitive help in as much or as little depth as they require. By linking chunks of help information in logical ways, you can develop a help system that is responsive to users' immediate needs.

Use the help system to inform users about short-cuts.

Short-cuts are facts that experts typically know. A help system that volunteers answers without forcing users to ask questions can help novices become experts.

Suggested Readings

Apple Computer, Inc. (1987). Human Interface Guidelines: *The Apple Desktop Interface*. Reading, MA: Addison-Wesley Publishing Co.

Borenstein, N.S. (1985). *The Design and Evaluation of On-line Help Systems.* Ph.d thesis, Carnegie-Mellon University.

Christensen, M. (1984). *Background for the Design of an Expert Consulting System for On-line Help.* Thesis proposal, Temple University.

Owen, D. (1986). *Answers first, then questions.* In D.A. Norman & S.W. Draper (Eds.), User Centered System Design, (p. 361-375). Hillsdale, NJ: Erlbaum.

Human Interface Notes

Note #3 **Dueling Metaphors: the Desktop & HyperCard**

Written by: Tom Erickson January 1990
(Supersedes Human Interface Update #14)

Discussion of the differences between the metaphors of the Desktop and HyperCard.

Metaphors help users form a coherent model of an application's human interface. In an interface with a well-chosen metaphor like the Desktop, users find it easy to predict the results of an action or to figure out which action produces a desired result.

While a single, clear metaphor aids human-computer communication, mixing metaphors may cause significant problems. Even two metaphors which work well separately may interfere with one another when they are used within the same human interface.

The Desktop and HyperCard metaphors can interfere with one another. This document describes the conditions under which such interference can occur, and what can be done to avoid it.

In the Desktop metaphor users do things by pressing rounded-rectangle buttons, choosing menu commands, and double-clicking (opening) icons. Each type of control object has a distinct, carefully-defined appearance, as well as a different method of access. You can tell how to use a control object just by looking at it.

In HyperCard, buttons are the principle control objects, but in HyperCard, a button can look like anything—an icon, an item in a list, a push button, a menu item. Since a click is the only way of starting an action, the appearance of a button is less important than in the Desktop: the user knows that all HyperCard control objects respond to a single click.

When elements of the Desktop and HyperCard metaphors are combined, confusion may result. In an interface with a mixed metaphor, a user can no longer predict the result of clicking an item in a list or clicking on an icon. Does a click select the object, as in the Desktop metaphor, or does it launch an action, as in the HyperCard metaphor? Clicking on an icon to select it and having it launch an action because it's acting like a HyperCard button is—at the very best—disconcerting. Such unpredictability destroys the comfortable feel that is essential to a good human interface[1]. Users confronted with such unpredictability are likely to become lost, confused, and unhappy with your product.

Do **not** mix the Desktop and HyperCard metaphors. If you're writing a HyperCard stack, don't include icon-like buttons that must be double-clicked. If you're writing a Desktop application, don't include (to take a real example) a house icon that takes the user somewhere when it's clicked, like the Home Card button in HyperCard. Desktop applications should not contain HyperCard-like interface elements.

The most important point is this: It should always be obvious whether the user is in a Desktop application or a HyperCard stack. And this means obvious at a glance; users should not have to read text or remember whether an application or a stack was launched. If the context is obvious, the user knows the result of a click—without having to think about it.

[1] Also see Chapter 1 of *Human Interface Guidelines: The Apple Desktop Interface* (Addison-Wesley, 1987)—in particular, the principles of Consistency and Perceived Stability.

Human Interface Notes

Note #4 Movable Modal Dialog Boxes

Written by: Scott Jenson January 1990

Discussion of a new modal window style that can be moved by dragging its title bar.

A standard modal dialog box works well as long as you, the developer, are asking such questions as "How do you want to print this document?" or "Save changes before quitting?" However, sometimes you need to ask a question and the user needs to see the document contents to make a decision. A common example is a Find… or Replace… dialog box. The usual rule of choice is to use a modeless dialog box since 1) it's less intrusive on the many different ways people may want to use your software, and 2) since it's movable, the user can easily move it around to view covered parts of the document. There are some cases, however, when the question or response task needs to be modal, but the user still might want to view what's behind the dialog box. An example would be a complex attribute change like adding a border to a paragraph of text. You might want to see the text or even other paragraphs while you're setting up the border.

In these cases use a movable modal dialog box. This window design gives you visual feedback both that it is a modal dialog box and also that you can drag it from the title bar. Figure 1 shows a simple example of this dialog box style.

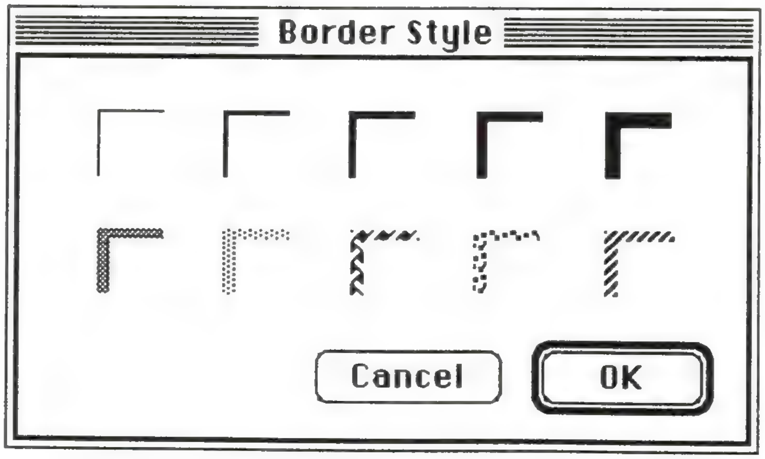


Figure 1–Movable modal dialog box

A couple of points to keep in mind:

- Any selection made in the dialog box should immediately update the document contents. The OK button then means “accept this change” and the Cancel button means “undo all changes done by this dialog box.” Some applications use an Apply button to approximate this behavior but this only confuses the meaning of OK and Cancel.
- With this dialog box, it is not necessary to keep your application from switching to other MultiFinder layers. System 7.0 uses this method to show an application is busy with some time-consuming operation, yet can still be switched into the background.
- When you create this dialog box, be sure to use the new window type. Do **not** draw a `rect` in a `documentProc`. System 7.0 has as new selector on the standard `'WDEF'` resource for this type of window. For System Software 6.0.x, you can obtain a new `'WDEF'` resource on AppleLink in the the Human Interface section of the Developer Services Bulletin Board or request a copy by writing to AppleLink address MACINTERFACE.
- Make sure to save the position of the window for the next time it’s used.
- Do **not** use this dialog box when a modeless dialog box would work instead.

Human Interface Notes

Note #5 **What "Cancel" Means**

Written by: John Sullivan January 1990

"Cancel" means "dismiss this operation, with no side effects." It does not mean "done with the dialog box," "stop what you are doing no matter what," or anything else.

When to use Cancel

In alert or dialog boxes, use Cancel for the name of a button that closes the alert or dialog box and returns the system to the state it was in before the alert or dialog box was displayed. When a lengthy operation is in progress, use Cancel for the name of a button that dismisses the operation and returns the machine to the state it was in before the operation began, with no side effects.

What to do the rest of the time

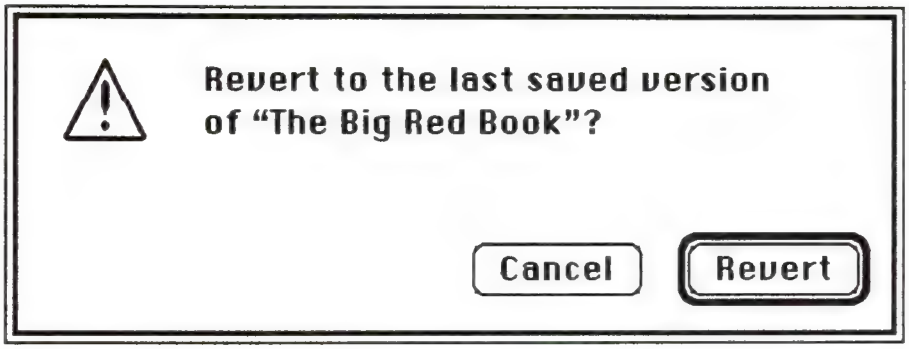
When it is impossible to return to the state that existed before an operation began, do not use the word Cancel. Two common alternatives, useful in different situations, are OK and Stop.

In alert or dialog boxes, use OK for the name of a button that closes the alert or dialog box and accepts any changes made while it was displayed. For confirmation alerts (alerts that say, essentially, "Are you sure you want to do this?") and many simple dialog boxes, it is better to use a word or two that succinctly describes what accepting the alert or dialog box means, such as Revert or Change All.

When a lengthy operation is in progress, use Stop for the name of a button that halts the operation before its normal completion, accepting the possible side effects. Stop may leave the results of partially-completed tasks around, but Cancel never does.

Some correct examples

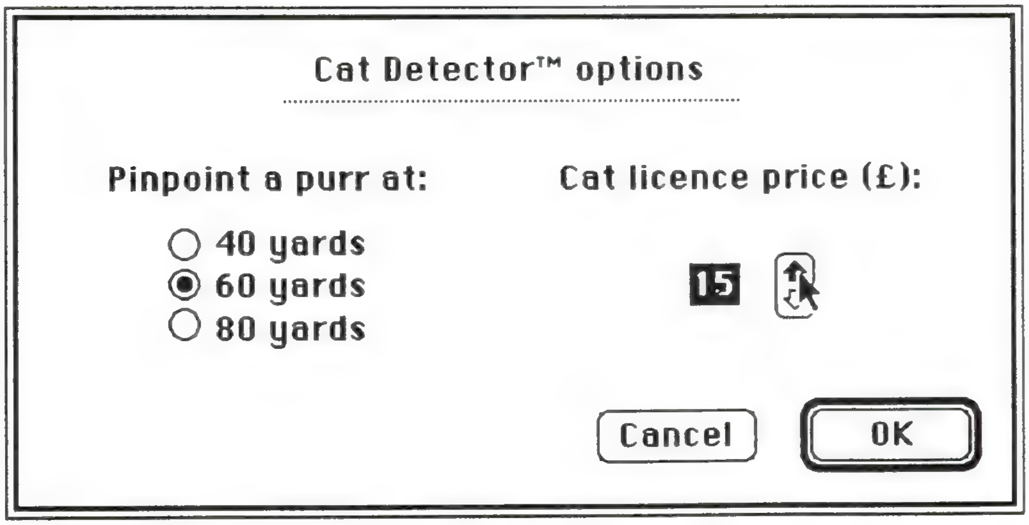
Following is a series of examples of proper uses of Cancel and its cousins.



Revert to the last saved version of “The Big Red Book”?

Figure 1–Confirmation alert

The alert in Figure 1 uses Revert instead of OK, since Revert neatly sums up what accepting the alert means.



Cat Detector™ options

Pinpoint a purr at:

○ 40 yards
● 60 yards
○ 80 yards

Cat licence price (£): 15

Figure 2–More complex dialog with Cancel and OK

The dialog box in Figure 2 uses OK because there is no succinct term to describe what accepting the changes in the dialog box means.

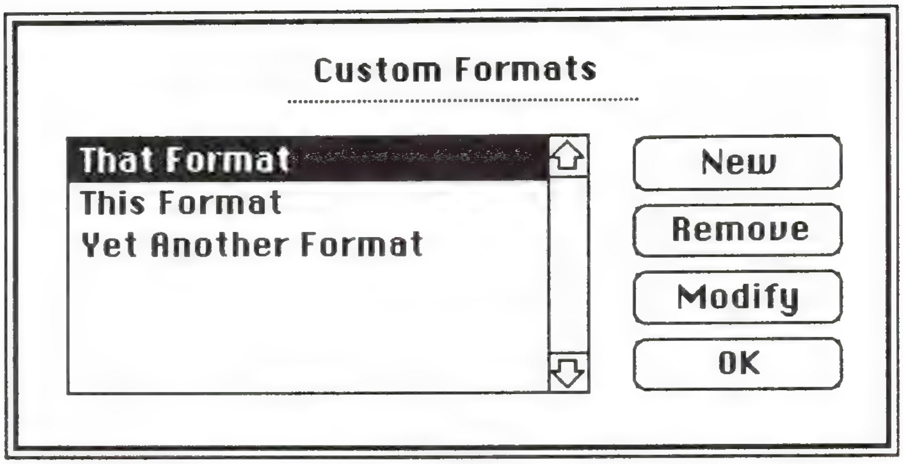


Figure 3–More complex dialog with OK instead of Cancel

The dialog box in Figure 3 uses OK because it doesn't throw away all the changes that were made since the dialog box was first drawn. If the button were named Cancel instead, clicking it would remove any formats created since the dialog box was drawn, bring back any formats removed since the dialog box was drawn, and undo any changes that had been made by selecting a format and clicking Modify since the dialog box was drawn.

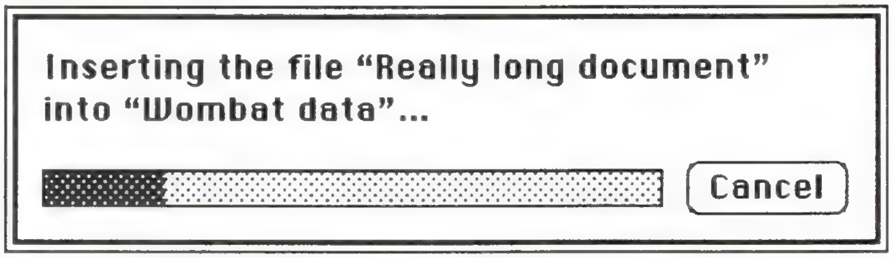
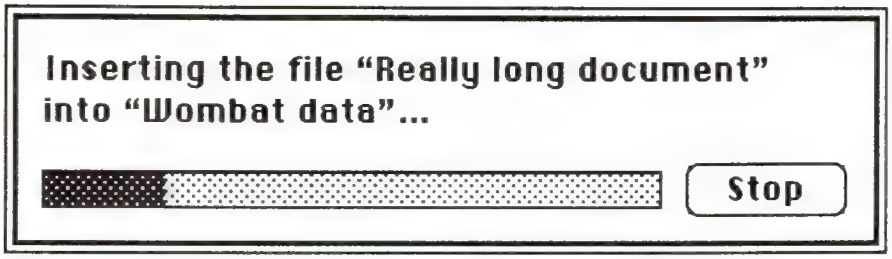

Inserting the file "Really long document" into "Wombat data"...

Figure 4–Progress indicator that uses Cancel

The dialog box in Figure 4 uses Cancel because clicking the button leaves the document named Wombat Data in the state it was in before the Insert File command was chosen.



Inserting the file “Really long document” into “Wombat data”...

Figure 5–Progress indicator that uses Stop

The dialog box in Figure 5 uses Stop because clicking the button stops inserting text into the document named Wombat Data, but it doesn’t remove the text that has already been inserted.

Human Interface Notes

Note #6 Window Positions

Written by: John Sullivan January 1990

Whenever a window is displayed on the screen, the application must make a decision about its size and location. If a user moved the window in an earlier session, then it should be restored to its previous position; otherwise, the application must choose an appropriate default position. This document gives details about making these decisions.

Saving and restoring window positions

Users change the locations and sizes of windows for a reason. They might want to view two documents side by side, or they might want to display a window on a larger monitor so that more of it can be seen at once. They might want to display only the interesting area of the window, which may be quite small, or they might want to position the window in such a way that certain icons on the desktop are still visible. In any case, one of the most important principles of a good interface is that the user is in control, so applications must respect the user's reasoning by reopening each window in the same location and with the same size that the user left it.

Here is a simple, but effective, procedure for saving and restoring window positions:

1. When opening a new window, put it in the default position (see the next section of this document for details about determining the default position).

2. Before closing a movable window, check to see if its location or size have changed. If so, save the new location and size. If the window can be zoomed, save the user state and also save whether or not the window is in the zoomed (standard) state. Note that if the window corresponds to a Finder document and there were no other changes to the document, the new location and size should be saved without changing the modification date of the document.

   Note: If the location and size of a movable window have not changed, do not save them, because the default location and size may be different the next time the window is opened (e.g., if the window is reopened on a different Macintosh with a different screen size).

3. When reopening a movable window, check its saved position. If the window is in a position to which the user could have dragged it, then leave it there. If the window can be zoomed and was in the zoomed state when it

was last closed, put it in the zoomed state again. (Note that the current and previous zoomed states are not necessarily the same, since the window may be reopened on a different monitor.) If the window is not in a position to which the user could have dragged it, then it must be relocated, so use the default location. However, do not automatically use the default size when using the default location; if the entire window would be visible using the default location and the stored size, then use the stored size.

Remember that checking to see if the saved position is reasonable before reopening the window is a necessary part of this procedure, not an option. When an application opens windows outside of the visible space, users tend to switch to competitors' products.

Choosing a default window position

The appropriate default position of a window may depend upon several factors, including whether the window is a document window or an alert, the locations of other open windows, the user's center of attention, and whether or not the window contains information that is closely related to other open windows. The rest of this document includes a series of default window position examples for several common cases. Developers should consider how their particular situations relate to these common ones to determine the best default positioning. In any case, the default position of any window must never overlap multiple screens, as this can look and feel very strange with monitors of different depth and resolution.

Independent document windows, single screen

On a single screen the first document window should be positioned in the upper-left region of the gray area of the screen. Each additional window should be staggered slightly below and to the right of its predecessor, if no windows are moved or closed. When a window is moved or closed, its original position becomes available again. The next window opened should use this position. Similarly, if a window is moved onto a previously-available position, that position becomes unavailable again. Figure 1 illustrates independent document windows on a single screen.

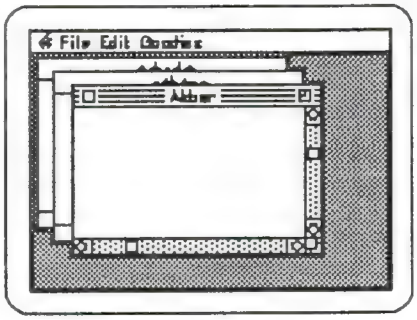


Figure 1–Independent document window positions

Independent document windows, multiple screens

The first document window should be positioned in the upper-left region of the gray area of the main screen (the screen with the menu bar). Each additional independent window should be staggered from the upper-left of the screen that contains the largest portion of the frontmost window. Thus, if the user starts the application, creates a single window, drags that window over to a secondary monitor, and then creates a second window, the second window and subsequent windows should appear on the secondary monitor. Figure 2 illustrates independent document windows on multiple screens.

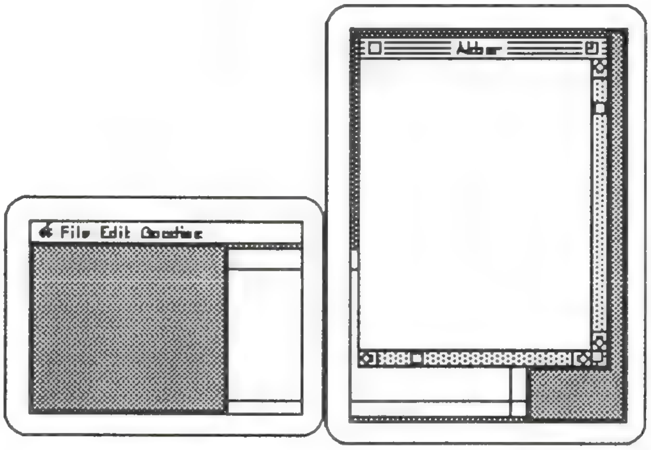


Figure 2–Independent document window positions on multiple screens

Child windows

A child window is a window that contains more detail about part of another window. For instance, in ResEdit a window showing all string resources of a given file is a child of the window showing all resource types for that file. Child windows of visible parent windows should be created just below and to the right of the parent window. Figure 3 illustrates child windows.

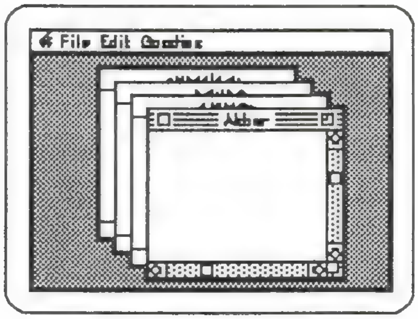
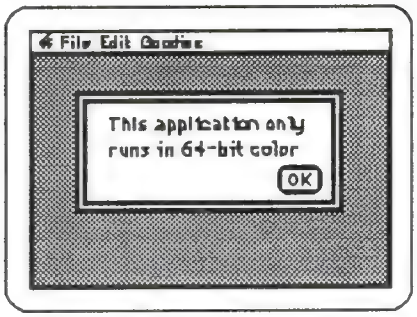


Figure 3–Child Windows

Alert or dialog box, single screen

Alerts or dialog boxes should be centered horizontally and positioned vertically such that one-third of the remaining vertical gray screen space is above the window and the other two-thirds are below. Figure 4 shows a typical alert.

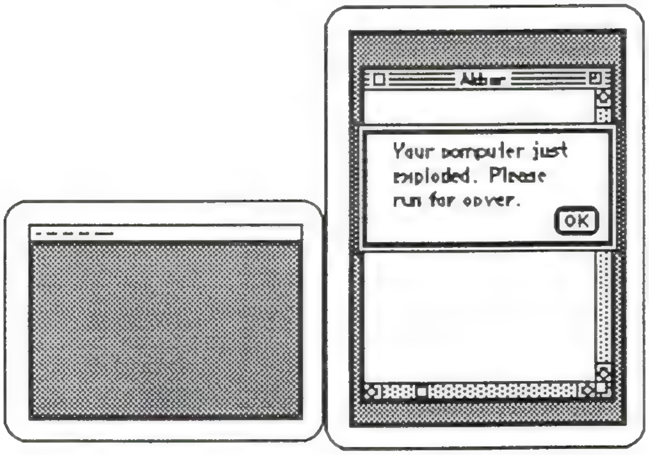


This application only runs in 64-bit color

Figure 4–Alert

Alert or dialog box, multiple screens

This case is similar to the previous one, except that the alert or dialog box should be drawn on the screen closest to the user's center of attention. Always putting an alert or dialog box on the screen containing the cursor is a good rule of thumb. An even better rule is to use the screen on which the last user action took place. For instance, if the user is typing into a word processing document and presses Command-O, put the standard file dialog box on the same screen as the word processing document. When an alert or dialog box appears in response to the user selecting a menu item with the mouse, put the alert or dialog box on the screen with the menu bar. Figure 5 shows an alert in a two-screen configuration.



Your computer just exploded. Please run for cover.

Figure 5–Alert on multiple screens

Human Interface Notes

Note #7 **Who's Zooming Whom?**

John Sullivan April 1990

Further discussion about using the zoom box.

Introduction

A click in the zoom box toggles a window between two states, the user state and the standard state. The user state, as its name implies, is set by the user. The standard state is defined by the Apple Human Interface Guidelines (p. 48) as "generally the full screen, or close to it...the size and location best suited to working on the document." That brief description has proven to be too brief in these days of larger and multiple monitors. This note is a more explicit guide to determining the appropriate standard state.

Size of the Standard State

When the zoom box was introduced, all Macintoshes had the same relatively small screen, so the "most useful" size of a window was almost always larger than the screen. Setting the standard state to the full screen size was, therefore, a good rule of thumb. This is no longer the case. These days, Macintosh monitors come in all shapes, sizes, and configurations, so applications should never simply assume that the standard state should be as large as the screen. Frequently the monitor is larger, sometimes much larger, than the most useful size for a window. Screen real estate is valuable, so screen-sized windows should be used only when they make sense.

For example, a document for a word processor has a well-defined "most useful width" (the width of a page) and a variable "most useful height" (depending on the number of pages). Therefore, the width of the standard state should be the width of a page or the width of the screen, whichever is smaller, and the height of the standard state should be the height of the screen or the height of the document, whichever is smaller.

Another example is a paint application whose documents are always exactly one page in size. In this case, the width of the standard state should be the width of a page or the width of the screen, whichever is smaller, and the height of the standard state should be the height of a page or the height of the screen, whichever is smaller.

Yet another example is an application that displays pictures but does not let users edit them. Since its pictures cannot be modified, making a window larger than the pictures it displays would not be useful. Therefore, the width of the standard state should be the width of the picture or the width of the screen, whichever is smaller, and the height of the standard state should be the height of the picture or the height of the screen, whichever is smaller. Note that this means that different document windows from the same application may have different standard states.

Position of the Standard State

One of the basic principles of the Apple Desktop Interface is "perceived stability." Users are more comfortable in an environment that does not change in an apparently random manner; a window need not move just because it is changing in size. When toggling a window from the user state to the standard state, first determine the appropriate size of the standard state. If this size would fit completely on the screen without moving the upper-left corner of the window, keep this corner anchored. Otherwise, move the window to an appropriate default location (see Human Interface Note #6, Window Positions).

The Standard State on Multiple Monitors

Zooming behavior in multiple monitor environments should not violate any of the guidelines described herein, but it does introduce a single additional rule: the standard state should be on the monitor containing the largest portion of the window, not necessarily on the monitor with the menu bar. Note that this means the standard state for a single window may be on different monitors at different times if the user moves the window around. In any case, the standard state for any window must always be fully contained on a single screen.

Further Reference

- Macintosh Technical Note #79, _ZoomWindow

Human Interface Notes

Note #8 Keyboard Equivalents

Revised by: Scott Jenson — June 1990
Written by: Scott Jenson — April 1990

A discussion of the standard Apple Human Interface keyboard equivalents.

Standard Keyboard Equivalents

N	New	**Z**	Undo
O	Open…	**X**	Cut
W	Close	**C**	Copy
S	Save	**V**	Paste
P	Print…	**A**	Select All
Q	Quit		

These keyboard equivalents are reserved across all applications. If your application does not support one of these commands, it should not use these keys for any other function. This restriction is for the user's benefit; it gives them guaranteed, predictable behavior across all applications. Using Command-O to mean "Open..." ninety-nine percent of the time and "Ostracize..." in your special case does two things: 1) users do not consider using Command-O, as it is already taken by all other applications, and 2) the variability of the equivalent only weakens their perception of consistency.

Other Common Keyboard Equivalents

F	Find…	**T**	Plain Text
G	Find Again	**B**	Bold
		I	Italic
		U	Underline

These Command keys equivalents are secondary to the standard keys previously listed. If your product does not support one of these functions, then feel free to use these equivalents as you wish.

Note that the keyboard equivalents for Print… and Plain Text are different from past Human Interface guidelines, which suggested P for Plain Text and nothing at all for Print. The marketplace has, by and large, standardized upon P for Print, leaving no common Command key equivalent for Plain Text. Apple has accepted this change and now suggests standardizing on T for Plain Text, based upon its mnemonic value and common usage among applications that use P for Print.

Unnecessary Command Keys

There should not be Command key equivalents for infrequently used menu commands. This type of usage only burdens your users and constrains your life even more. Only add Command key equivalents to commands your users use most frequently. As infrequently as it is chosen by most users, “Page Setup…” is an example of menu command that does not need a key equivalent.

Human Interface Notes

Note #9 **Pop-Up Menus**

Written by: Scott Jenson & John Sullivan June 1990

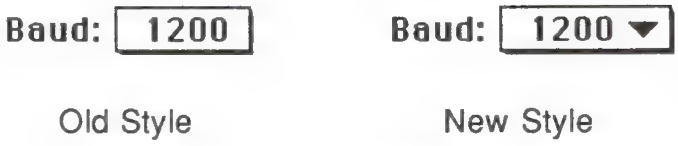
A description of the new style of pop-up menus.

Introduction

Pop-up menus have been around on the Macintosh since HFS (Hierarchical File System) was introduced in 1986, and their use became much more widespread after the addition of Toolbox support in 1987. It is surprising, then, that many Macintosh users have no idea what pop-up menus are and do not recognize them when they see them. The problem is that pop-up menus do not look sufficiently different from other Macintosh interface elements; the one-pixel drop shadow that differentiates pop-up menus from editable text fields has proven inadequate. This Note presents the new standard appearance for the pop-up menu in System Software 7.0 and also describes how the new appearance lends itself to some new uses that were previously impossible.

Standard pop-up menus

Previously, pop-up menus were displayed by surrounding the current value of the menu with a one-pixel rectangle and a one-pixel drop shadow to the right and bottom. The new standard appearance adds a downward-pointing black arrow, which is identical to the arrow that indicates that a menu is too long to fit on the screen and must scroll. All pop-up menus should now use this new style. Figure 1 shows a simple pop-up menu in both the old and new styles.



Old Style New Style

Figure 1–Old-style and new-style pop-up menus

Figure 2 shows an expanded view of the downward-pointing black arrow of this new-style pop-up menu.

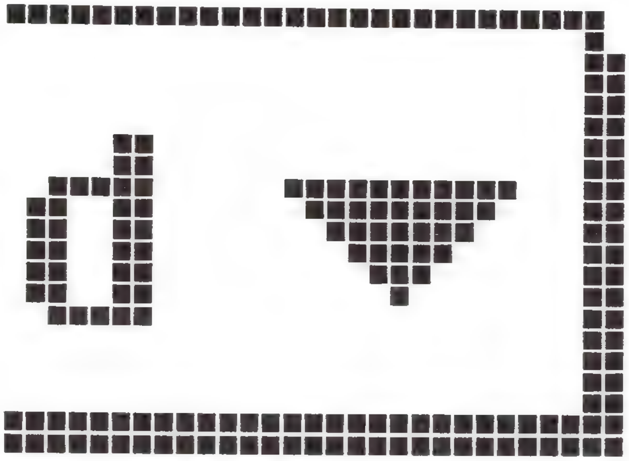


Figure 2–FatBits view of new-style pop-up menu

When the user clicks on the pop-up menu or its label text, the black arrow disappears and the menu pops up, and when the user releases the mouse button, the menu disappears and the black arrow is redrawn. Figure 3 illustrates the proper behavior of a pop-up menu when a user clicks on it.

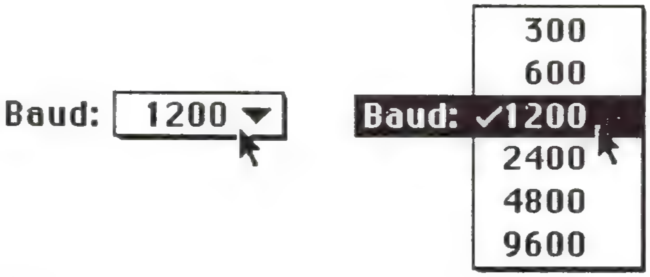


Figure 3–Pop-up menu before and during a mouse click

Pop-up menus with editable text fields

Sometimes it is useful to display a list of choices but still allow a user to enter or edit a choice that the application may not know in advance. One example is a font size field with an accompanying pop-up menu of commonly used sizes. The new standard pop-up menu appearance leads itself readily to this use, as shown in Figure 4.

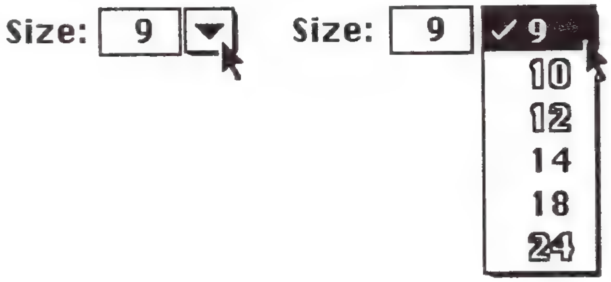


Figure 4–Pop-up menu with an editable text field

Note that as in standard pop-up menus, the black arrow disappears when a user clicks on it and reappears when a user releases the mouse button. Also note that an application should draw the pop-up menu so it automatically highlights the item that corresponds to the value in the edit text field; this technique prevents a quick click in the pop-up menu from accidently erasing the previous value.

If a user enters a value in the edit text field that does not match any of the pop-up menu's items, then the pop-up menu should make that value the first item and separate it from the rest of the standard values with a gray line., as shown in Figure 5. This separation makes a clean distinction between common items, which are always available, and the user-entered value, which is only temporary. (In the case of the example in Figure 5, if the font size 13 had been inserted in order into the list, a subsequent selection of 10, or any other matching selection, would have removed it from the list.)

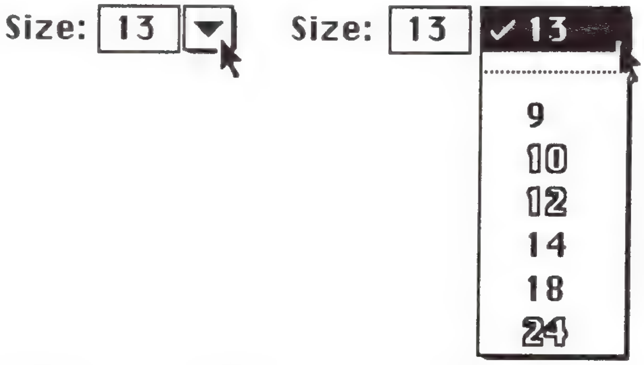


Figure 5–Pop-up menu with a non-matching edit text item

Human Interface]|[Notes

Note #10 **Alert Box Guidelines**

Written by: John Sullivan
(Slightly plagiarized from Kate Gomoll)

June 1990

Some simple rules to follow for alert boxes.

Why there are alert box guidelines

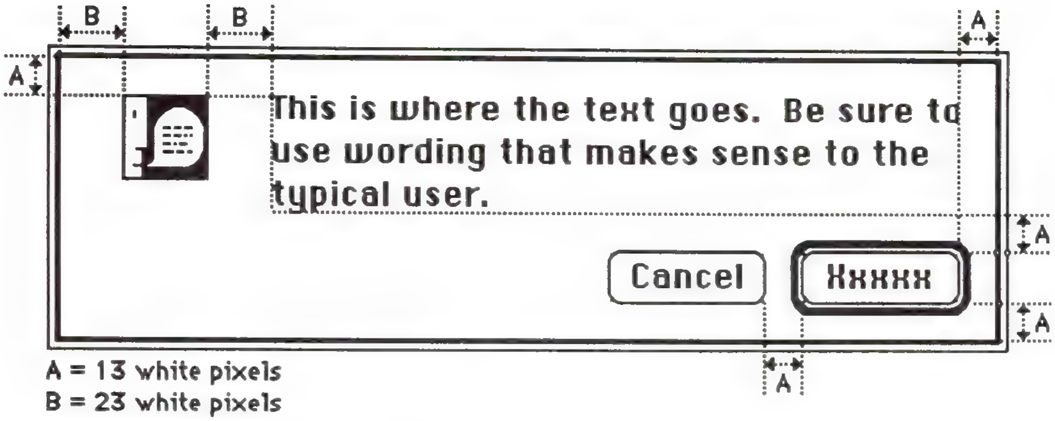
From the feedback the Human Interface group has received, it is clear that the discussion of alert boxes in *Human Interface Guidelines: The Apple Desktop Interface* is both imperfect and incomplete.

These guidelines serve at least three major purposes. First, they provide a simple recipe for making attractive alert boxes. Second, they provide a simple recipe (the same one, in fact) for making alert boxes that have a standard appearance and behavior. This standardization is important, because the more familiar the appearance of an alert box is to users, the easier it is for them to concentrate on the specific message being communicated. Finally, these guidelines provide simple rules that can be extended for designing more complicated dialog boxes.

This Note supplements and partially replaces the discussion in *Human Interface Guidelines: The Apple Desktop Interface*, so where this Note and the book disagree, believe this Note.

Alert box layout

Alert boxes drawn with the toolbox calls `_StopAlert`, `_NoteAlert`, and `_CautionAlert` place the icon in the rectangle (top = 10, left = 20, bottom = 42, right = 52); however, placement of all other elements is left to the individual designer. Figure 1 shows a simple alert box in which spacing between elements is based upon this placement of the icon.



This is where the text goes. Be sure to use wording that makes sense to the typical user.

Figure 1–Simple alert box with spacing

Following are the exact coordinates used in Figure 1 and how they were derived, in Rez format. Note that there are three white pixels built into the dialog frame and that the upper left corner of the text item is not the same as the upper left corner of the first character.

The first set of definitions are not actual coordinates, but instead are intermediate values used to derive them:

```
#define A                    13     // white space between most elements
#define B                    23     // white space to left and right of icon
#define NumTextLines         3      // number of lines of text in the alert
#define LineHeight           16     // height of a single line of Chicago-12
#define ButtonHeight         20     // standard button height
#define LongestButtonName    41     // width of "Cancel" in Chicago-12
#define ButtonWidth          59     // (LongestButtonName + 18)
```

The rest of the definitions are actual coordinates defining the window size (`AlertWidth` and `AlertHeight`) and the icon, text, and button locations:

```
#define AlertWidth           341    // chosen to make the right margin = A

#define IconLeft             20     // (B - 3)
#define IconRight            52     // (IconLeft + 32)
#define IconTop              10     // (A - 3)
#define IconBottom           42     // (IconTop + 32)

#define TextLeft             74     // (IconRight + (B - 1))
#define TextRight            331    // (AlertWidth - (A - 3))
#define TextTop              7      // (A - 6)
#define TextBottom           55     // (TextTop + (NumTextLines * LineHeight))

#define ButtonTop            68     // (TextBottom + A)
#define ButtonBottom         88     // (ButtonTop + ButtonHeight)
#define ActionButtonRight    331    // (AlertWidth - (A - 3))
#define ActionButtonLeft     272    // (ActionButtonRight - ButtonWidth)
#define CancelButtonRight    259    // (ActionButtonLeft - A)
#define CancelButtonLeft     200    // (CancelButtonRight - ButtonWidth)

#define AlertHeight          98     // (ButtonBottom + (A - 3))
```

The action button

Alert boxes that provide the user a choice should be worded as questions to which there is an unambiguous, affirmative response. The button for this affirmative response is called the action button. Whenever possible, label the action button with the action that it performs. Button names such as Save, Quit, or Erase Disk allow experienced users to click the correct button without reading the text of a familiar dialog. These labels are often clearer than words like OK or Yes. Phrase the question to match the action that the user is trying to perform. For instance, if the user selects Revert to Saved, the confirmation alert should say something like "Revert to the last saved version of the document? Any changes made since the last save will be lost." This message is much clearer than something like "Discard changes made since the last save?"

If the action cannot be condensed conveniently into a word or two, use OK. Also use OK when the alert is simply giving the user information without providing any choices.

The cancel button

Whenever possible, provide a button that allows the user to back out of the operation that caused the alert box to be displayed. This button is activated when the user types Command-. (period) or presses the Escape key. (Note that the Command key sequence may differ depending upon the script system in use. See Macintosh Technical Note #263, International Canceling, for more information.) Apple recommends naming this button Cancel, so that users can easily identify it as the safe escape hatch. For more information, see Human Interface Note #5, "What Cancel Means."

The default button

In most cases, the default button should perform the most likely action (if that can be determined). This usually means completing the action that the user initiated to display the alert box in the first place; therefore, the default button is usually the same as the action button. The default button is boldly outlined, and its action is performed when the user presses the Return or Enter key.

If the most likely action is dangerous (for example, it erases the hard disk), the default should be a safe button, typically the cancel button. If none of the choices are dangerous and there is not a likely choice, then there should be no default button.

When there is no default button, the user must explicitly click on one of the buttons (pressing Return or Enter does not perform an action). By requiring users to explicitly click on a button, you can protect them from accidentally damaging their work by pressing the Return or Enter key out of habit.

Buttons (placement, size, capitalization, and feedback)

Put the action button in the lower right corner, with the cancel button to its left. Use this placement regardless of which button is the default button; put the action button in the lower right corner even if the cancel button is the default.

Buttons in alert boxes look best when they are 20 pixels high (not counting the default button outline) and have at least 8 white pixels on either side of each button's name. These specifications mean that the width of the button should be at least 18 pixels larger than the width of the longest button name (16 pixels for the white space plus 2 pixels for the edges). It looks best to make all buttons the same width, unless the buttons' names have extremely different length names. If you find yourself tempted to make buttons with extremely long names, reconsider the names carefully; button names should be simple, concise, and unambiguous.

Capitalize the first letter of each button name, but never capitalize the entire name—with the single exception of the OK button. The OK button should always be named OK and never ok, Ok, Okay, okay, OKAY, or any even stranger variation. If a button name contains more than one word, capitalize each word, such as Replace All or Cancel Printing.

As in all dialog boxes, any buttons that are activated by key sequences must flash to give visual feedback as to which item has been chosen. A good rule of thumb is to invert the button for eight ticks; this is long enough so that it is always visible, but short enough that it is not annoying. Alert box calls in the Toolbox use the eight tick value by default.

Further Reference

- Human Interface Note #5, What Cancel Means
- Macintosh Technical Note #263, International Canceling

#0: About Macintosh Sample Code February 1990

Technical Note #0 (this document) accompanies each release of Macintosh Sample Code. This release includes revisions to Sample Code #11, #13-#14 and new Sample Code #19-#22 (originally dated October 1989 on *Phil & Dave's Excellent CD: The Release Version*). If there are any subjects which you would like to see treated in Sample Code (or if you have any questions about existing Sample Code), please contact us at one of the following addresses:

Macintosh Sample Code
Developer Technical Support
Apple Computer, Inc.
20525 Mariani Avenue, M/S 75-3T
Cupertino, CA 95014
AppleLink: MacDTS
MCI Mail: MacDTS
Internet: MacDTS@AppleLink.Apple.com

We want Sample Code to be distributed as widely as possible, so they are sent to all Partners and Associates at no charge; they are also posted on AppleLink in the Developer Services bulletin board and other electronic sources, including the Apple FTP site (IP 130.43.2.2). You can also order them through APDA. As an APDA customer, you have access to the tools and documentation necessary to develop Apple-compatible products. For more information about APDA, contact:

APDA
Apple Computer, Inc.
20525 Mariani Avenue, M/S 33-G
Cupertino, CA 95014
(800) 282-APDA or (800) 282-2732
Fax: (408) 562-3971
Telex: 171-576
AppleLink: APDA

We place no restrictions on copying Sample Code, with the exception that you cannot resell them, so read, enjoy, and share. We hope Sample Code will provide you with lots of valuable programming techniques while you are developing Macintosh hardware and software. These examples have undergone extensive review by DTS and Apple engineering, so we feel that the quality of the code is very high. However, it is likely that there are still some bugs that we have overlooked (unintentionally, of course), so we would appreciate hearing from you if you find any. If you are the first to report a particular bug, you will be the recipient of a genuine DTS kudo.

The following pages list all the Macintosh Sample Code that has been released:

Released Macintosh Sample Code February 1990

	Number	Title	Languages	New *** Revised *R* Release Date
	1	Sample	C,P,A	6/89
	2	TESample	C,P,a,X	6/89
	3	SillyBalls	C,P	8/88
	4	TubeTest	C,P	8/88
	5	HierMenus	P	8/88
	6	PopMenus	P	8/88
	7	FracApp	P	8/88
	8	FracAppPalette	P	8/88
	9	FracApp300	P	8/88
	10	EditCdev	C,P	8/88
R	11	GetZoneList	C,P,a	2/90
	12	Signals	C,P,A	11/88
R	13	OOPTESample	OOP,a	2/90
R	14	CPlusTESample	C++,a	2/90
	15	Offscreen	P	4/89
	16	OffSample	P,a	4/89
	17	TbltDrvr	A	4/89
	18	StdFile	C,P	4/89
***	19	TEStyleSample	P	2/90
***	20	Transformer	OOP	2/90
***	21	ModalList	C	2/90
***	22	ScreenFKey	P,a	2/90

Key to Languages

A	Assembly language version
P	Pascal language version
C	C language version
OOP	Object-Oriented Pascal
C++	C++
X	Can compile under A/UX
a	Some assembly required
b	batteries included

Macintosh
Sample Code Notes

Developer Technical Support

#1: Sample

Written by: Darin Adler, Mark Bennett, and Jim Reekes

Versions:	1.00	August 1988
	1.01	November 1988
	1.02	April 1989
	1.03	June 1989

Components:	`Sample.p`	June 1, 1989
	`Sample.c`	June 1, 1989
	`Sample.a`	June 1, 1989
	`Sample.inc1.a`	June 1, 1989
	`SampleMisc.a`	June 1, 1989
	`Sample.r`	June 1, 1989
	`Sample.h`	June 1, 1989
	`PSample.make`	June 1, 1989
	`CSample.make`	June 1, 1989
	`ASample.make`	June 1, 1989

Major changes since 1.0

Revamped the way that memory availability is checked and handled at initialization. Substantially changed the way windows are closed. Added an error message dialog to better inform users, and improved error handling in general. Finally, put a funny hack into the C version so we could call `_PurgeSpace` under MPW 2.0.2.

Search for "1.01" in the code to find all the specific changes.

Major changes since 1.01

Removed all dependencies on MPW 2.0; this version requires MPW 3.0 or later. Improved `TrapAvailable` to handle differences between machines prior to the Macintosh II and the Macintosh II and later models.

Search for "1.02" in the code to find all the specific changes.

Sample is an example application that demonstrates how to initialize the commonly used Toolbox managers, operate successfully under MultiFinder, handle desk accessories, and create, grow, and zoom windows.

It does not, by any means, demonstrate all the techniques you need for a large application. In particular, Sample does not cover exception handling, multiple windows or documents, sophisticated memory management, printing, or undo, all of which are vital parts of a normal full-sized application.

This application is an example of the form of a Macintosh application; it is **not** a template. It is **not** intended to be used as a foundation for the next world-class, best-selling, 600K application. A stick figure drawing of the human body may be a good example of the form for a painting, but that does not mean it should be used as the basis for the next *Mona Lisa*.

We recommend that you review this program or TESample before beginning a new application.

Macintosh Sample Code Notes

Developer Technical Support

#2: TESample

Written by: Mark Bennett, Rick Blair, and Dave Radcliffe

Versions:	1.00		August 1988
	1.01		November 1988
	1.02		April 1989
	1.03		June 1989
Components:	`TESample.p`		June 1, 1989
	`TESample.c`		June 1, 1989
	`TESampleGlue.a`	•	June 1, 1989
	`TESample.r`		June 1, 1989
	`TESample.h`		June 1, 1989
	`PTESample.make`	•	June 1, 1989
	`CTESample.make`	•	June 1, 1989
	`TESampleGlue.s`	••	June 1, 1989
	`TESampleAUX.r`	••	June 1, 1989
	`Makefile`	••	June 1, 1989

MPW Only • A/UX Only ••

Major changes since 1.0
Revamped the way that memory availability is checked and handled at initialization. Substantially changed the way windows are closed. Added an error message dialog to better inform users, and improved error handling in general. Finally, put a funny hack into the C version so we could call `_PurgeSpace` under MPW 2.0.2.

Search for "1.01" in the code to find all the specific changes.

Major changes since 1.01
Removed all dependencies on MPW 2.0; this version requires MPW 3.0 or later. Improved `TrapAvailable` to handle differences between machines prior to the Macintosh II and the Macintosh II and later models.

A/UX programmers
Version 1.02 introduces conditionals for compilation under A/UX 1.1. Note that the binary file compiled under MPW will run fine under A/UX. These changes were made to provide an example of how to produce source files which can be compiled under both MPW and A/UX.

Search for "1.02" in the code to find all the specific changes.

TESample is an example application that demonstrates how to initialize the commonly used Toolbox managers, operate successfully under MultiFinder, handle desk accessories, and create, grow, and zoom windows. It demonstrates fundamental TextEdit toolbox calls and TextEdit automatic scrolling, and it shows how to create and maintain scroll bar controls.

It does not, by any means, demonstrate all the techniques you need for a large application. In particular, TESample does not cover exception handling, multiple windows or documents, sophisticated memory management, printing, or undo, all of which are vital parts of a normal full-sized application.

This application is an example of the form of a Macintosh application; it is **not** a template. It is **not** intended to be used as a foundation for the next world-class, best-selling, 600K application. A stick figure drawing of the human body may be a good example of the form for a painting, but that does not mean it should be used as the basis for the next *Mona Lisa*.

We recommend that you review this program or Sample before beginning a new application. Sample is a simple application which does not use TextEdit or the Control Manager.

Macintosh
Sample Code Notes

Developer Technical Support

#3: SillyBalls

Written by: Bo3b Johnson

Versions:	1.00	August 1988
Components:	`SillyBalls.p`	August 1, 1988
	`SillyBalls.c`	August 1, 1988
	`PSillyBalls.make`	August 1, 1988
	`CSillyBalls.make`	August 1, 1988

SillyBalls is a very simple application that demonstrates how to use Color QuickDraw. It is about two pages of code, and it does nothing more than open a color window and draw randomly colored ovals in the window.

The purpose of SillyBalls is to demonstrate how to get quick results with Color QuickDraw. It is a complete program, but it is very short to be as clear as possible. It does not have an event loop, and it is not fully functional, in the sense that it does not do all the things one would normally expect from a well-behaved Macintosh application (i.e., use an event loop, size the window naturally, use menus, etc.)

We recommend that you review Sample or TESample for the general structure and MultiFinder techniques you should use when writing a new application.

#4: TubeTest

Written by:	Bo3b Johnson	
Versions:	1.00	August 1988
Components:	`TubeTest.p`	August 1, 1988
	`TubeTest.c`	August 1, 1988
	`TubeTest.r`	August 1, 1988
	`PTubeTest.make`	August 1, 1988
	`CTubeTest.make`	August 1, 1988

TubeTest is a very simple demonstration of how to use the Palette Manager in a color application. It has a special color palette that is associated with the main window, and the colors are animated using the Palette Manager to give a flowing tube effect. The program is very simple; the Palette Manager and drawing routines are in separate subroutines to make it easier to figure out what is happening.

TubeTest is a complete program with a main event loop (MEL), so there is extra code to run in the MEL. A resource file is necessary to define the menu, window, dialog, and palette resources which the program uses.

We recommend that you review Sample or TESample for the general structure and MultiFinder techniques you should use when writing a new application.

Macintosh
Sample Code Notes

Developer Technical Support

#5: HierMenus

Written by: Bryan Stearns

Versions:	1.00	August 1988
Components:	`HierMenus.p`	August 1, 1988
	`HierMenus.r`	August 1, 1988
	`HierMenus.make`	August 1, 1988

HierMenus is a very simple demonstration of how to use hierarchical menus in your application.

We recommend that you review Sample or TESample for the general structure and MultiFinder techniques you should use when writing a new application.

Macintosh Sample Code Notes

Developer Technical Support

#6: PopMenus

Written by: Bryan Stearns

Versions:	1.00	August 1988
Components:	`PopMenus.p`	August 1, 1988
	`PopMenus.r`	August 1, 1988
	`PopMenus.make`	August 1, 1988

PopMenus is a very simple demonstration of how to use pop-up menus in your application. It implements a pop-up menu as a `userItem` in a modal dialog box (this is a helpful example in its own right).

We recommend that you review Sample or TESample for the general structure and MultiFinder techniques you should use when writing a new application.

Macintosh
Sample Code Notes

Developer Technical Support

#7: FracApp

Written by: Bo3b Johnson

Versions:	1.00	August 1988
Components:	`MFracApp.p`	August 1, 1988
	`UFracApp.p`	August 1, 1988
	`UFracApp.incl.p`	August 1, 1988
	`FracApp.r`	August 1, 1988
	`FracApp.make`	August 1, 1988

This program requires MPW 2.0.2 and MacApp 1.1.1 to build.

This is the "commercial quality" version of FracApp. This version handles multiple documents, and it supports color table animation using an off-screen `gDevice` with a port. The updates to the screen using `_CopyBits` are as fast as possible. FracApp does not use the Palette Manager, except to provide for the system palette, or color modes with less than 255 colors. For the color table animation, it uses the Color Manager and handles the colors itself. Strict compatibility was relaxed to allow for a higher performance program. This is the most "real" of the FracApp programs.

As color on the Macintosh evolves, we hope that future versions of this program will be able to use the Palette Manager and maintain the level of performance. To achieve this, we will have to attain better QuickDraw (i.e., `_CopyBits`) and Palette Manager integration.

#8: FracAppPalette

Written by: Bo3b Johnson

Versions:	1.00	August 1988
Components:	`MFracAppPalette.p`	August 1, 1988
	`UFracAppPalette.p`	August 1, 1988
	`UFracAppPalette.incl.p`	August 1, 1988
	`FracAppPalette.r`	August 1, 1988
	`FracAppPalette.make`	August 1, 1988

This program requires MPW 2.0.2 and MacApp 1.1.1 to build.

This version of FracApp uses the Palette Manager. It demonstrates a full-color palette which is used to display the Mandelbrot set. FracAppPalette does not support color table animation, since the integration of QuickDraw (i.e., `_CopyBits`) and the Palette Manager is not yet full enough.

This version uses an off-screen `gDevice` with a port to handle the data, using `_CopyBits` to draw into the window. The palette is automatically associated with each window. The PICT files are read and written using the bottlenecks (spooled) to save on memory usage.

Macintosh
Sample Code Notes

Developer Technical Support

#9: FracApp300

Written by: Bo3b Johnson

Versions:	1.00	August 1988
Components:	`MFracApp300.p`	August 1, 1988
	`UFracApp300.p`	August 1, 1988
	`UFracApp300.incl.p`	August 1, 1988
	`FracApp300.r`	August 1, 1988
	`FracApp300.make`	August 1, 1988

This program requires MPW 2.0.2 and MacApp 1.1.1 to build.

This version of FracApp does not support colors, but it does demonstrate how to create and use a 300 dpi bitmap with a port. The bitmap is printed at full resolution on LaserWriter printers and clipped on other printers (but it still prints). FracApp300 demonstrates how to use a high-resolution image as a PICT file and how to print it.

#10: EditCdev

Written by: Mark Bennett

Versions:	1.00	August 1988
Components:	`EditCdev.p`	August 1, 1988
	`EditCdev.c`	August 1, 1988
	`EditCdev.r`	August 1, 1988
	`CEditCdev.make`	August 1, 1988
	`PEditCdev.make`	August 1, 1988

EditCdev is a sample Control Panel device (cdev) that demonstrates the use of the edit-related messages and how to implement an `editText` item in a cdev. It utilizes the new undo, cut, copy, paste, and delete messages that are sent to cdevs in response to user menu selections.

EditCdev is comprised of two `editText` items which can be edited and selected with the mouse or the Tab key.

Macintosh Sample Code Notes

Developer Technical Support

#11: GetZoneList

Written by: Mark Bennett & Pete Helme

Versions:	1.00	November 1988
	1.1	February 1990
Components:	`GetZoneList.p`	February 1, 1990
	`GetZoneList.c`	February 1, 1990
	`GetZoneList.r`	February 1, 1990
	`PGetZoneList.make`	February 1, 1990
	`CGetZoneList.make`	February 1, 1990
Required:	`UFailure.p`	November 1, 1988
	`UFailure.incl.p`	November 1, 1988
	`UFailure.a`	November 1, 1988

GetZoneList is a sample application that uses AppleTalk's AppleTalk Transaction Protocol (ATP) and Zone Information Protocol (ZIP) to obtain a list of zones on an AppleTalk internet. It also demonstrates using a signal, or failure-catching mechanism, to recover from error situations.

GetZoneList is based on Sample, and we recommend that you review Sample or TESample for the general structure and MultiFinder techniques you should use when writing a new application.

Macintosh
Sample Code Notes

Developer Technical Support

#12: Signals

Written by: Rick Blair

Versions:	1.00	Technical Note #88 and MacApp 1.1
	2.00	November 1988
Components:	`UFailure.p`	November 1, 1988
	`UFailure.h`	November 1, 1988
	`UFailure.incl.p`	November 1, 1988
	`UFailure.a`	November 1, 1988
	`TestSignal.p`	November 1, 1988
	`TestCignal.c`	November 1, 1988
	`TestSignal.make`	November 1, 1988
	`TestCignal.make`	November 1, 1988

UFailure (or Signals) is a set of exception handling routines suitable for use with MacApp, MPW Pascal, and MPW C. It is a “jazzed-up” version of the original MacApp UFailure unit, and it includes a set of C interfaces too.

#13: OOPTESample

Written by: Keith Rollin

Versions:	1.00	April 1989
	1.1	February 1990
Components:	`BuildOOPTESample`	February 1, 1990
	`MOOPTESample.p`	February 1, 1990
	`OOPTESample.make`	February 1, 1990
	`TECommon.h`	February 1, 1990
	`TESampleGlue.a`	February 1, 1990
	`TESample.r`	February 1, 1990
	`UApplication.p`	February 1, 1990
	`UApplication.inc1.p`	February 1, 1990
	`UDocument.p`	February 1, 1990
	`UDocument.inc1.p`	February 1, 1990
	`UTEDocument.p`	February 1, 1990
	`UTEDocument.inc1.p`	February 1, 1990
	`UTESample.p`	February 1, 1990
	`UTESample.inc1.p`	February 1, 1990

The build process for OOPTESample is entirely automated. All you need to do is run the BuildOOPTESample script. BuildOOPTESample is a variation on the BuildProgram script that comes standard with MPW. It creates a folder to contain the intermediary object files, and then calls Make with the file OOPTESample.make. Make's output is executed with the final application OOPTESample as the result.

OOPTESample is an example application that demonstrates how to initialize the commonly used Toolbox managers, operate successfully under MultiFinder, handle desk accessories, and create, grow, and zoom windows. It demonstrates fundamental TextEdit toolbox calls and TextEdit automatic scrolling, and it shows how to create and maintain scroll bar controls.

This version of TESample has been substantially reworked in Object Pascal to show how a "typical" object-oriented program could be written. To this end, what was once a single source code file has been restructured into a set of classes which demonstrate the advantages of object-oriented programming.

There are four main classes in this program. Each one of these has an interface (.p) file and an implementation (.inc1.p) file, and is compiled into its own separate `UNIT`.

The TApplication class does all of the basic event handling and initialization necessary for Macintosh Toolbox applications. It maintains a list of TDocument objects and passes events to the correct TDocument class when appropriate.

The TDocument class does all of the basic document handling work. TDocuments are objects that are associated with a window. Methods are provided to deal with update, activate, mouse-click, key-down, and other events. Some additional classes which implement a linked list of TDocument objects are provided.

The TApplication and TDocument classes together define a basic framework for Macintosh applications, without having any specific knowledge about the type of data being displayed by the application's documents. They are a (very) crude implementation of the MacApp application model, without the sophisticated view hierarchies or any real error handling.

The TESample class is a subclass of TApplication. It overrides several TApplication methods, including those for handling menu commands and cursor adjustment, and it does some necessary initialization. Note that we only need to override nine methods to create a useful application class.

The TEDocument class is a subclass of TDocument. This class contains most of the special-purpose code for text editing. In addition to overriding most of the TDocument methods, it defines a number of additional methods which are used by the TESample class to get information on the document state.

This program consists of four segments. "Main" contains most of the code, including the MPW libraries and the main program. "Initialize" contains code that is used only once, or rarely, and can be unloaded after the event loop is completed. "%A5Init" is automatically created by the Linker to initialize globals for the MPW libraries and is unloaded right away. "%_MethTables" is a fake segment used by Object Pascal to maintain object relationships.

Toolbox routines do not change the current port. In spite of this, in this program we use a strategy of calling `_SetPort` whenever we want to draw or make calls which depend on the current port. This precaution makes us less vulnerable to bugs in other software which might alter the current port (such as the bug (feature?) in many desk accessories which changes the port when there is a call to `_OpenDeskAcc`). Hopefully, this also makes the routines from this program more self-contained, since they don't depend on the current port setting.

This program does not maintain a private scrap. Whenever a cut, copy, or paste occurs, we import or export from the public scrap to TextEdit's scrap right away, using the `TEToScrap` and `TEFromScrap` routines. If we did use a private scrap, the import or export would be in the activate or deactivate event and suspend or resume event routines.

Macintosh
Sample Code Notes

Developer Technical Support

#14: CPlusTESample

Written by: Andrew Shebanow

Versions:	1.00	April 1989
	1.1	July 1989
	1.2	February 1990
Components:	`CPlusTESample.make`	February 1, 1990
	`TApplicationCommon.h`	February 1, 1990
	`TApplication.h`	February 1, 1990
	`TDocument.h`	February 1, 1990
	`TECommon.h`	February 1, 1990
	`TESample.h`	February 1, 1990
	`TEDocument.h`	February 1, 1990
	`TApplication.cp`	February 1, 1990
	`TDocument.cp`	February 1, 1990
	`TESample.cp`	February 1, 1990
	`TEDocument.cp`	February 1, 1990
	`TESampleGlue.a`	February 1, 1990
	`TApplication.r`	February 1, 1990
	`TESample.r`	February 1, 1990

CPlusTESample is an example application that demonstrates how to initialize the commonly used Toolbox managers, operate successfully under MultiFinder, handle desk accessories, and create, grow, and zoom windows. It demonstrates fundamental TextEdit toolbox calls and TextEdit automatic scrolling, and it shows how to create and maintain scroll bar controls.

This version of TESample has been substantially reworked in C++ to show how a "typical" object-oriented program could be written. To this end, what was once a single source code file has been restructured into a set of classes which demonstrate the advantages of object-oriented programming.

There are four main classes in this program. Each one of these has a definition (.h) file and an implementation (.cp) file.

The TApplication class does all of the basic event handling and initialization necessary for Macintosh Toolbox applications. It maintains a list of TDocument objects and passes events to the correct TDocument class when appropriate.

The TDocument class does all of the basic document handling work. TDocuments are objects that are associated with a window. Methods are provided to deal with update, activate, mouse-click, key-down, and other events. Some additional classes which implement a linked list of TDocument objects are provided.

The TApplication and TDocument classes together define a basic framework for Macintosh applications, without having any specific knowledge about the type of data being displayed by the application's documents. They are a (very) crude implementation of the MacApp application model, without the sophisticated view hierarchies or any real error handling.

The TESample class is a subclass of TApplication. It overrides several TApplication methods, including those for handling menu commands and cursor adjustment, and it does some necessary initialization. Note that we only need to override nine methods to create a useful application class.

The TEDocument class is a subclass of TDocument. This class contains most of the special-purpose code for text editing. In addition to overriding most of the TDocument methods, it defines a number of additional methods which are used by the TESample class to get information on the document state.

Macintosh
Sample Code Notes

Developer Technical Support

#15: Offscreen

Written by: Rick Blair

Versions:	1.00	April 1989
Components:	`Offscreen.p` `Offscreen.incl.p`	April 1, 1989 April 1, 1989

These routines provide a high-level interface to the QuickDraw and Color Manager routines which allow the creation and manipulation of off-screen bitmaps and pixel maps. They are designed to run on any machine with 128K or later ROMs (sorry 64K ROM fans).

Note that the design incorporates the idea that you can go along pretending there is an off-screen buffer, even when one could not be allocated, and the calls will do nothing.

#16: OffSample

Written by: Mark Bennett

Versions:	1.00	April 1989
Components:	`OffSample.p`	April 1, 1989
	`OffSample.r`	April 1, 1989
	`OffSample.h`	April 1, 1989
	`POffSample.make`	April 1, 1989
Required:	`Offscreen.p`	April 1, 1989
	`Offscreen.incl.p`	April 1, 1989
	`UFailure.p`	November 1, 1988
	`UFailure.incl.p`	November 1, 1988
	`UFailure.a`	November 1, 1988

OffSample demonstrates the usage of the Offscreen unit. It shows how to use off-screen bitmaps and pixel maps to produce flicker-free updating with a minimum of code restructuring. OffSample attempts to reduce the amount of “knowledge” it has of the off-screen structure so as to minimize its dependence upon that unit.

OffSample emphasizes using the Offscreen unit; it is not intended to be viewed as a complete application on which to base some larger effort. Instead, its method of using off-screen bitmaps and pixel maps should be studied and adapted to other applications that desire features like flicker-free updating.

#17: TbltDrvr

Written by: Cameron Birse

Versions:	1.00	April 1989
Components:	`TbltDrvr.a`	April 1, 1989

`'ADBS'` resources are loaded and executed at boot time (before INIT 31), and they are made of two main parts, the installation or initialization code and the the actual driver.

In this example, the installation portion allocates memory in the system heap for the service routine and the "optional data area." It installs the driver using the Apple Desktop Bus (ADB) Manager call `_SetADBInfo`.

Generally speaking, ADB devices are intended to be user input devices. The ADB Manager polls the bus every 11 milliseconds to see if a device has new user input data. This polling is accomplished by sending a talk R0 command to the last active device. The last active device is the last device that had data to send to the host. If another device has data, it can request a poll by sending a service request signal to the host.

When a device has responded to a poll, the ADB Manager will call the driver to process the data. This call is done a interrupt time (level 1), and the driver is passed the data, by getting a pointer to a Pascal string which contains the actual data.

In this example, the data is in the form of a pointing device's coordinates and button state. When the driver gets the data, it stores the coordinate information in `RawMouse` and `MTemp`. We stuff both `RawMouse` and `MTemp`, because the tablet is an "absolute" device. It also checks the state of the button against `MBState`, and if there has been a change, it will update `MBState` and post either a mouse-up or mouse-down event, as appropriate.

Note: This code demonstrates how to move the cursor position. This information is meant for input device drivers only; this technique should not be used by applications to move the cursor. Moving the cursor is bad user interface, and nobody likes a bad user interface, so "Just Say No."

#18: StdFile

Written by: Keith Rollin

Versions:	1.00	April 1989
Components:	`StdFile.c`	April 1, 1989
	`StdFile.p`	April 1, 1989
	`StdFile.h`	April 1, 1989
	`StdFile.r`	April 1, 1989
	`StdFile.rsrc`	April 1, 1989
	`CStdFile.make`	April 1, 1989
	`PStdFile.make`	April 1, 1989

StdFile attempts to demonstrate the following techniques:

- Normal use of `SFGetFile` and `SFPutFile`.
- Normal use of `SFPGetFile` and `SFPPutFile`, which includes the use of custom dialogs and handling extra items through the implementation of a `DlgHook`.
    - First time initialization.
    - Extra simple buttons (Quit, Directory, ThisDir).
    - Radio buttons (file format, types of files to show).
    - Aliasing—click on some buttons to click on other buttons.
    - Regenerating the list of files displayed.
- Creating a full pathname from a reply record (using working directories or `DirID`)
- Selecting a directory (*à la* MPWs “GetFileName -d”)
- Simple file filter (checks file type).
- Complex file filter (looking inside the file).
- Adding and deleting List Manager lists and extra List Manager lists. This is shown for both `SFGetFile` and `SFPutFile`.
- Select multiple files using one of two methods.
    - Replace StdFile’s list with one of your own.
    - Add a second list to the dialog box. This method is not shown explicitly, but rather, I show how to install and dispose of the actual list item. Inserting filenames into the list is left as an exercise to you, the programmer.
- Setting the starting directory or volume.
- Describe pending update event clogging.

Note: This application assumes the existence of HFS. It makes HFS calls and accesses HFS data structures without first checking to see if HFS exists on this machine.
In some cases, you will see me make use of a peculiar Pascal syntax: `IF <expr> THEN;`. This is intentional, as it gets the Pascal compiler to discard function results in which I have no interest.

Macintosh
Sample Code Notes

Developer Technical Support

#19: TEStyleSample

Written by: Mary Burke

Versions:	1.00	February 1990
Components:	`PTEStyleSample.p`	February 1, 1990
	`TEStyleSampleGlue.a`	February 1, 1990
	`TEStyleSample.r`	February 1, 1990
	`TEStyleSample.h`	February 1, 1990
	`PTEStyleSample.make`	February 1, 1990

TEStyleSample is an example application that demonstrates how to initialize the commonly used toolbox managers, operate successfully under MultiFinder, handle desk accessories and create, grow, and zoom windows. Both styled and fundamental TextEdit toolbox calls and TextEdit auto-scroll are demonstrated. It also shows how to create and maintain scroll bar controls as well as implement a basic printing loop.

Macintosh
Sample Code Notes

Developer Technical Support

#20: Transformer

Written by: Keith Rollin

Versions:	1.00	February 1990
Components:	`MTransformer.p`	February 1, 1990
	`Transformer.c`	February 1, 1990
	`Transformer.r`	February 1, 1990
	`UTransformer.p`	February 1, 1990
	`UTransformer.incl.p`	February 1, 1990
	`Transformer.MAMake`	February 1, 1990
Additional Documentation:	*The Amazing Bitmap Transmogrifier*	

Transformer is a sample program that demonstrates:

- bitmap transformations
- mixing MacApp with C subroutines
- mixing 68881 and non-68881 code together
- calling of MacApp routines from C
- using CursorCtl routines
- turning on and off the MacApp BusyCursor mechanism

It uses a MacApp shell to open file, open windows, and handle menus, but uses a core routine written in vanilla C to perform the actual transformation. The transformation consists of translating, scaling, and rotating. The comments in the source code are sparse, if existant at all, so gleaning how the transformation routine works is very difficult. To explain what is going on, a sister document, *The BitMap Transmogrifier*, has been included. It explains all of the necessary math, and shows how the formulas were derived. There are also lots of pictures.

Adding C routines to a MacApp program used to be a pain, but no longer. Previously, weird gyrations had to be performed in order to get things to link correctly and without lots of warnings or errors. Now with MacApp 2.0ß9 and later, support has been explicitly provided in MABuild for mixing in C.

For best performance, the C routine is compiled with the `-mc68881` option (this is set in the MAMake file). A problem arises with this, as an extended value is passed from the non-FPU Pascal code to the FPU C code. Since the size of extended values changes depending on the setting of the 68881 options, the parameters have to be converted as per page 347 of the MPW 3.0 Pascal manual.

In our C routine, we make use of some of the MacApp utilities. This is done by making a small set of external declarations that match the Pascal interfaces for the routines we are interested in. This is done for FailNIL, FailOSErr, and BusyActivate.

BusyActivate is a routine that controls the BusyCursor mechanism of MacApp. This mechanism gives MacApp programs a built-in watch cursor that kicks in whenever the application is involved in a lengthy process. During our transformation routine we want this turned off, as we supply our own busy indicator with the CursorCtl library routines.

The CursorCtl library routines allow one to implement the spinning beachball cursor. We set this up when we initialize our application with a call to InitCursorCtl. This reads in our `'acur'` and `'CURS'` resources and initializes them in whatever way it deems necessary. When we need to show the spinning cursor, we just start calling SpinCursor() with some rotation value. This rotation value is added to an internal counter. When this counter reaches 32, the next cursor specified in the `'acur'` resource is shown. More information on this is included in the interface files for CursorCtl.

#21: ModalList

Written by: James Beninghaus

Versions:	1.00	February 1990
Components:	`ModalList.c`	February 1, 1990
	`ModalList.h`	February 1, 1990
	`ModalList.r`	February 1, 1990
	`ModalList.make`	February 1, 1990

ModalList is an example using a list in a dialog window. The default LDEF is used to display a two-dimensional list of strings. You can scroll the list, search for and change cell contents, and change the list's selection flags.

Please review Sample or TESample for the general application structure and MultiFinder techniques you should use when writing a new application. This sample is meant to demonstrate the use of the List Manager and Dialog Manager routines.

Macintosh
Sample Code Notes

Developer Technical Support

#22: ScreenFKey

Written by: Guillermo Ortiz

Versions:	1.00	February 1990
Components:	`ScreenFKey.p`	February 1, 1990
	`ScreenFKey.a`	February 1, 1990
	`ScreenFKey.make`	February 1, 1990

ScreenFKey is a basic example on how to spool a PICT file to disk by replacing the bottleneck PutPICProc, it saves the contents of the screen to a file. The FKEY creates ten files Screen 0 through Screen 9; it is necessary to erase or rename old files when the limit is reached.

This FKEY works in any Macintosh computer and saves the screen regardless of the setting of the screen; to use, it has to be added to the System file using ResEdit.

ATTENTION 32-BIT QD USERS: A 'feature' in 32 Bit Quickdraw makes it possible for this FKEY to fail under normal conditions (normal means it would work okay without 32-Bit QD) the symptoms are lost cursor and empty pictures. The cheap solutions are use Finder or increase the partition size for your application. As a quick reference if the main screen is set to 8-bits deep then 32-Bit QD will try to allocate a 300K handle, when the main screen is a direct device 32-bit deep then 1.2M is necessary. Future versions will compensate for this anomalous behavior.